National Library
of Canada

Canadian Theses Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale
du Canada

Service des thèses canadiennes

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

University of Alberta

An Interactive Floorplanner for Integrated Circuit Design

By

R. Scott Stephens ©

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science.

Department of Electrical Engineering

Edmonton, Alberta
Fall, 1991

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-70038-6

Canada

University of Alberta

Release Form

NAME OF AUTHOR: R. Scott Stephens

TITLE OF THESIS: An Interactive Floorplanner for Integrated Circuit Design

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1991

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

R. Scott Stephens

10627-42 St.
Edmonton, Alberta
Canada

Date October 10, 1991

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled An Interactive Floorplanner for Integrated Circuit Design submitted by R. Scott Stephens in partial fulfillment of the requirements for the degree of Master of Science.

Jack Mowchenko

Supervisor: Dr. Jack Mowchenko

Bruce Cockburn

Dr. Bruce Cockburn

Xiabo Li

Dr. Xiabo Li

Date October 7, 1991

Abstract

This thesis presents a new interactive floorplanning utility where floorplanning is based on a multi-level circuit hierarchy. A multi-level circuit hierarchy reduces the number of components which make up any module, which in turn reduces the size of the problem. All occurences of a module in a floorplan use the same module definition, further reducing the required floorplanning effort. The floorplanner allows a designer to create multiple alternative floorplans for each module and the floorplanner chooses the combination of alternatives which minimizes the total area consumption of the final circuit. This thesis discusses the advantages this approach has over other approaches, and discusses the implementation of these features within the utility. The performance of this floorplanner on a common circuit is discussed along with conclusions about the implementation and the trade-offs of features.

Table of Contents

1. Introduction 1
    - 1.1 Method of Circuit Design 1
        - 1.1.1 Methods of Circuit Layout 3
    - 1.2 Circuit Layout Approaches 5
        - 1.2.1 Macro-Cell Placement 5
        - 1.2.2 Floorplanning with Module Synthesis 6
    - 1.3 Merits of Layout Alternatives 7
    - 1.4 Merits of Layout Automation 8
    - 1.5 FLINT: A New Floorplanning Utility 8
    - 1.6 Thesis Outline 9
2. Previous Work 10
    - 2.1 Circuit Layout Methods 10
        - 2.1.1 Macro-Cell Placement 11
        - 2.1.2 Floorplanning with Module Generation 12
    - 2.2 Literature Survey 13
        - 2.2.1 Introduction to Layout Organizations 13
        - 2.2.2 Standard Cell Placement 15
            - 2.2.2.1 Standard Cell Placement Using Partitioning and Slicing. 15
            - 2.2.2.2 Standard Cell Placement Using Non-Slicing Structures... 19
        - 2.2.3 Macro-Cell Placement 21
        - 2.2.4 Floorplanning 28
            - 2.2.4.1 Floorplanning Using Slicing Structures 29

2.2.4.2 Floorplanning Using Non-Slicing Structures.................... 39
2.2.4.3 Comparison of Algorithmic Floorplanning Methods........ 40
2.2.4.4 Expert System Floorplanning Methods............................ 41
2.3 Inadequacies in Floorplanners.. 47
2.4 How Are These Inadequacies Addressed Here?.............................. 48
3. FLINT: A New Floorplanner.. 49
3.1 Definitions.. 49
3.2 Design Objectives.. 50
3.3 Overview of Features and Limitations... 52
3.4 Introduction to Slicing Structures... 55
3.5 Floorplanner Operations.. 57
3.5.1 Alternative Creation and Deletion.. 57
3.5.2 Circuit Viewing... 57
3.5.3 Operations Affecting Module Floorplans............................ 59
3.5.4 Operations on Iterated Structures.. 63
3.5.5 Operations Affecting Circuit Hierarchy.............................. 65
3.5.5.1 Operations Affecting Circuit Hierarchy and Placement... 67
3.6 Other Features Under Designer Control... 69
3.7 Features Under Floorplanner Control... 69
4. Implementation Details... 71
4.1 Data Structures.. 71
4.1.1 Hierarchy Tree... 71
4.1.1.1 Module and Instance Records.. 71
4.1.1.2 Slice Direction Records.. 73
4.1.1.3 Iteration Records.. 77

4.1.1.4 Other Fields........ 79
4.2 Algorithms........ 80
4.2.1 Support Routines........ 81
4.2.2 Operation Routines........ 83
4.3 Tools........ 102
4.3.1 Organized C........ 103
4.3.2 SunView Graphics........ 104
4.3.3 CapFast Schematic Editor........ 105
5. Results........ 107
5.1 Information About FLINT........ 107
5.2 Circuit Description........ 107
5.3 Steps Involved in Circuit Design........ 110
5.4 Floorplan Results........ 112
5.5 Performance of FLINT........ 114
5.5.1 Impressions About FLINT........ 114
5.5.2 Suggested Changes to FLINT........ 115
6. Conclusions........ 117
6.1 Synopsis........ 117
6.2 Success of FLINT........ 118
6.3 Performance Using Slicing Structures........ 120
6.4 Limitations of FLINT........ 120
6.5 Future Work........ 122
Appendix A. Block Diagram of AM2901........ 126
Appendix B. Floorplans of Test Circuits........ 127
Appendix C. Screen Image of Floorplan........ 130

List of Figures

Figure 1.1: Domains of Description 2

Figure 2.1: Slicing vs. Non-Slicing Structures 14

Figure 2.2: Block Partitioning 17

Figure 2.3: Channel Graph and Floorplan 22

Figure 2.4: Floorplan and Two Slicing Trees 36

Figure 2.5: Floorplans and Proper Polish Expressions 37

Figure 2.6: Planar Triangulated Graphs 43

Figure 2.7: Rectangular Duals 45

Figure 2.8: Four-Completion of Planar Triangulated Graph 45

Figure 3.1: Slicing Structure and Slicing Tree 56

Figure 3.2: Change Circuit View 58

Figure 3.3: Changing Slice Directions 60

Figure 3.4: Changing the Order of Children 60

Figure 3.5: Moving a Block to the End of a Row 61

Figure 3.7: Cut and Paste Within the Same Module 62

Figure 3.8: Splitting a Structure Followed by Changing a Slice Direction 64

Figure 3.9: Interleaving Iterated Structures 65

Figure 3.10: Flattening the Circuit Hierarchy 66

Figure 3.11: Cut and Paste Across Hierarchy Levels 68

Figure 4.1: Simple Floorplan and Tree 72

Figure 4.2: Two Level Floorplan and Tree 73

Figure 4.3: Floorplan of Two Blocks/Four Modules and Tree 74

Figure 4.4: Two Modules With Common Instance.. 75

Figure 4.5: Tree With Two Alternatives.. 76

Figure 4.6: Floorplan with Iterated Structure and Tree.. 78

Figure 4.7: Iterated Structure and Tree No Slice Node.. 78

Figure 4.8: Iterated Structure and Tree With Slice Record.. 79

Figure 4.9: Clean Up of Tree.. 82

Figure 4.10: Floorplan Before and After Flatten.. 86

Figure 4.11: Changing Slice Direction and Tree.. 90

Figure 4.12: Floorplan and Tree After New Slice Direction Change.. 90

Figure 4.13: Changing a Slice Between Blocks.. 91

Figure 4.14: Changing Slice Separating Two Components.. 91

Figure 4.15: Changing Slice Between Iterations.. 92

Figure 4.16: Changing Slice Which Separates Blocks of One Iteration.. 92

Figure 4.17: Steps of Clean Up After a Slice Direction Change.. 95

Figure 5.1: Change Slice Direction.. 115

Figure A.1: Block Diagram of AM2901, From Schematic 126

Figure B.1: Layout of Test One.. 127

Figure B.2: Layout of Test Two.. 128

Figure B.3: Layout of Test Three.. 129

Figure C.1: Screen Image of Final Floorplan of Test One.. 130

Chapter One: Introduction

The automation of circuit design is becoming more desirable as the demand for integrated circuits increases. This thesis introduces and discusses the development of a new floorplanning utility: FLINT (FLoorplanning INTeractively). This chapter discusses, in general terms, the paths which circuit design takes, concentrating on the role of physical layout. The first section of this chapter gives an overview of circuit design. The second section takes a look at physical circuit design, concentrating on two main approaches to circuit layout. Section 3 discusses the merits of the two main approaches presented in section 2. Section 4 discusses the merits of layout automation. Section 5 of this chapter presents an outline of this thesis.

1.1 Method of Circuit Design

There are two major activities involved in creating an integrated circuit (IC): design and fabrication. Design ends, and fabrication begins, with the description of a set of masks. This set of masks describes all the geometrical relationships between the components which make up the IC. The entire design process creates these masks starting with a very general description of circuit behavior. To design a circuit you must convert a circuit description from the behavioral domain to the structural domain and, in turn, convert the description from the structural domain to the physical domain.

These three domains describe the same circuit in different ways. The behavioral domain describes how a circuit acts. The structural domain describes the construction of a circuit, at an abstract level. The physical domain describes the actual implementation of the circuit, the location of components and the connections between them.

The behavioral, structural and physical domains of circuit description have several levels of detail. Walker and Thomas [WaTh85] have described a model of design representation which uses three axes, one each for the behavioral, structural, and physical representations. Figure 1.1 shows a diagram of this representation. Descriptions become progressively more detailed toward the centre of the diagram.

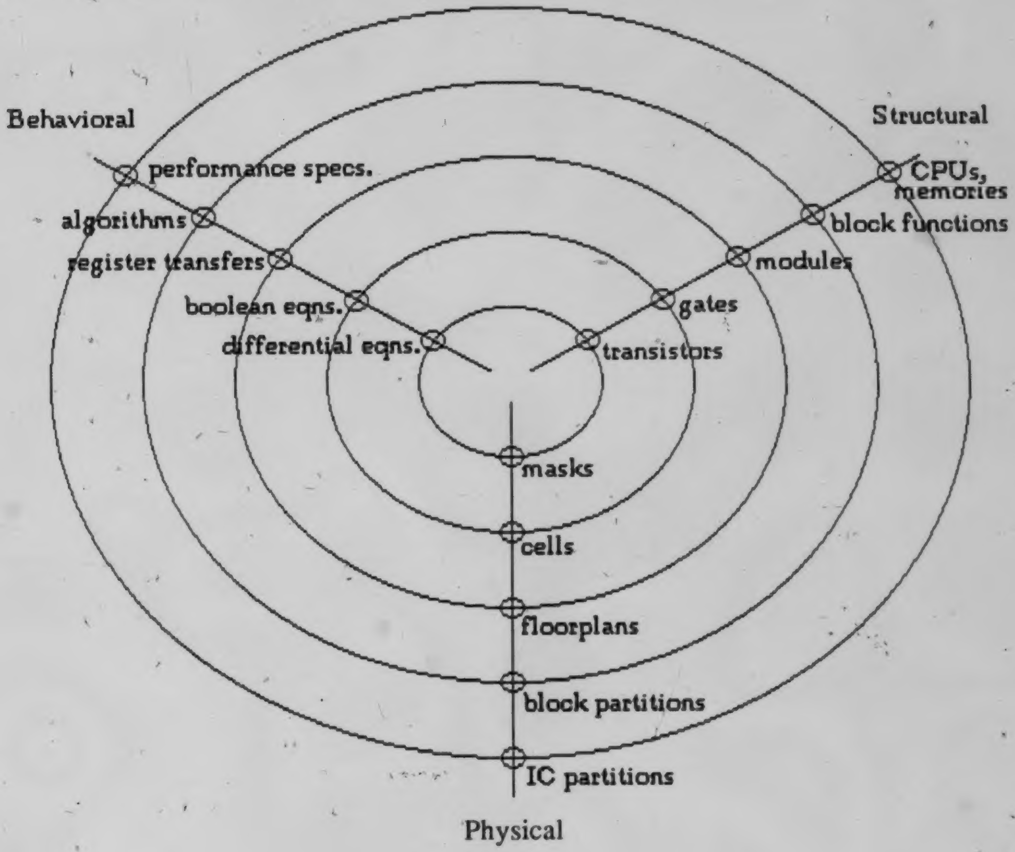

Behavioral
Structural
performance specs.
algorithms
register transfers
boolean eqns.
differential eqns.
CPUs, memories
block functions
modules
gates
transistors
masks
cells
floorplans
block partitions
IC partitions
Physical

Figure 1.1: Domains of Description

Converting a design from performance specifications in the behavioral domain to cell details in the physical domain does not require specifying the circuit at all levels in all domains during the design process. For example, the actual design path may convert performance specifications to algorithms, algorithms to hardware modules, hardware modules to arithmetic logic units (ALU), multiplexors (MUX) and registers, these to floorplans, floorplans to cells, and finally cells to masks. This is not the only path to follow when designing an IC. For example, although this path does not describe the circuit in terms of boolean equations, this information is implicit to the design, and the designers could recreate it if need be.

Automated tools may handle any number of tasks on a given design path. Some tools handle all the tasks, while others handle only one. FLINT is a floorplanner which converts circuits described in the structural domain to the floorplan level of the physical domain, and from there converts them to the module or cell level. This thesis is not involved with tasks in the behavioral domain. Tasks in the structural domain are only of interest here because they provide the inputs to FLINT. Before describing what a floorplanner is and what it does, we give an overview of physical circuit design.

1.1.1 Methods of Circuit Layout

Companies in the integrated circuit industry spend a great number of man-hours on the physical design of IC's. Physical design involves five broad steps, some of which have become routine. These steps are: 1) *Partitioning the circuit ~~into~~ ICs*. This involves deciding what functions each IC should perform, maximum size, power consumption, etc. 2) *Partitioning the ICs into functional blocks or modules*. The modules are sets of circuitry which perform a common task, or share a large number of connections. This step may also include restricting the power available to modules, and limiting their sizes and shapes. This step may also decide how each module will be implemented (such as programmed logic arrays (PLA) or standard cells). Modules at this level may be items such as ALUs, RAM files, or registers. 3) *Floorplanning the functional blocks or modules*. This is the physical placement of modules in relation to each other, determining their sizes and shapes. This step may also assign interconnects to routing channels between modules. 4) *Placing the gates within the modules*. This involves arranging the components which make up each module, or perhaps generating each module from a set of boolean equations (as may be done if the module is created as a PLA). This step also assigns the connections between components to routing channels. 5) *Creating masks from the gates*. This step involves describing the components with a series of masks. Indirectly these masks determine where the material which makes up each component will go. This includes metal for wiring, polysilicon for transistor gates and wiring, and diffusion regions for transistor sources and drains, etc.

The circuit design steps in the physical domain require input from the structural domain. Circuit partitioning cannot take place without a functional block description from the structural domain. Floorplanning cannot proceed without knowledge of module interconnections furnished from the structural domain. The floorplanner may obtain some of this knowledge from a library, hiding these steps during the current design process. Nonetheless, this information must be available in some form.

A tool capable of turning a circuit description from the structural domain into a set of masks is called a structural silicon compiler (SSC). Generally an SSC performs steps 2 through 5 in the physical domain described. Step 1 in the physical domain must be done prior to using the SSC.

1.2 Circuit Layout Approaches

Structural silicon compilers employ two common alternatives to handle steps 3 and 4. One of these is macro-cell placement, the other is floorplanning with module synthesis. Both approaches start with a circuit which is already partitioned and whose connections are known.

There are several objectives to consider when laying out circuits. These include minimizing the area which the circuit will consume, minimizing the length of interconnects, balancing the power consumption of various regions of the IC, and maximizing the speed of the final circuit. These goals are not always compatible and trade-offs must be made. In this thesis, the goal of minimizing the total area consumed by the final circuit is the only goal given much consideration in analysing the performance of layout utilities.

1.2.1 Macro-Cell Placement

Macro-cell placement arranges modules already described in the physical domain. These modules have a fixed shape and fixed external terminal connections. In fact, mask descriptions of each module could be generated before macro-cell placement begins. The job of macro-cell placement is to arrange these fixed modules next to each other, minimizing both the interconnect length and the total area consumed by the final circuit. The macro-cell placement utility can alter the relative placement of modules, orient modules on their sides, and mirror them horizontally and vertically. After the macro-cell placement tool decides on a final configuration, the modules must have their interconnections joined. The macrocell placement program may do this step or it may be done by a separate routing utility.

The modules used in macro-cell placement may come from a library of available modules, from a PLA generator or a gate matrix generator, or created in some other fashion. Modules not coming from a library are generated before macro-cell placement begins.

1.2.2 Floorplanning with Module Synthesis

Floorplanners are also concerned with arranging modules in the physical domain. However, floorplanners do not work exclusively with fixed shape modules. Most of the modules which floorplanners place are flexible modules which are fully created after floorplanning is complete. So, besides deciding on the placement of modules, the floorplanner must also decide on their areas and aspect ratios (the height to width ratio). The floorplanner does not have total control over a modules area and aspect ratio. Each module has restrictions on its shape and comes with some predefined relationship between area and aspect ratio with which the floorplanner must work. After the floorplanner creates the floorplan, a module generator creates the modules. The module generator arranges the components of the module in relation to each other and routes them together. The goals of a module generator are the same as the goals of floorplanners and macrocell placement utilities. The floorplanner gives the module generator some shape constraints with which it must work. These constraints may be that the module must have a certain aspect ratio, or not exceed a maximum height or width. There is no guarantee that the module generator will create modules which exactly fit the area and aspect ratio which the floorplanner expected. If some module grossly exceeds its expected area, the floorplanner may have to make changes to the floorplan. After the module generator creates the modules and the floorplanner places them, they must be interconnected. This final step may be done independently of the floorplanner.

The two step process of floorplanning and module generation can be extended to a multi-level hierarchical layout process. There have been some proposals for this kind of hierarchical floorplanning. The floorplanner must floorplan all levels in the hierarchy, except the lowest level. The module generator creates the flexible modules at the lowest level of the hierarchy. The hierarchical floorplanner which we have created is the subject of this thesis.

1.3 Merits of Layout Alternatives

Floorplanning has a greater potential to create good placements than macro-cell placement because module shapes can be chosen so that the modules fit together well. Macro-cell placement is a slave to the predefined shapes. Unless module dimensions are chosen carefully, they may not fit together well. Floorplanning has a second advantage because module generation takes place after floorplanning. Since the locations of modules is known, the locations of connections between modules are also known and this information can be used to optimize the location of each module's external terminal locations around the perimeter. This, in turn, reduces the total circuit wiring length, resulting in additional space savings.

Floorplanning with module generation does have some drawbacks. Since floorplanning manipulates more parameters than macro-cell placement, it is a more complex task, and it usually takes more computer time than macro-cell placement to produce equivalently high quality results. In addition, if the floorplanner does not predict module shapes accurately, unsatisfactory floorplans can result. Accurate shape functions can be difficult to generate.

1.4 Merits of Layout Automation

Circuit layout takes a long time to do manually. There are many layout options to choose from which may or may not provide an acceptable (though probably not ideal) solution. Manual floorplanning has associated with it a great amount of non-creative labour. Automated algorithms provide much faster solutions than manually generated ones, but generally result in poorer designs. Design aids, which handle most of the non-creative labour while leaving the creative part of floorplanning to a designer, have been virtually overlooked.

Despite the proliferation of integrated circuits, their design has remained the domain of large companies and research institutions. Circuit layout has remained inaccessible in large part because it is a complex and time consuming process. Tools which aid in the layout phase of circuit design are required to bring circuit design within the reach of smaller companies.

1.5 FLINT: A New Floorplanning Utility

In an effort to advance the state-of-the-art of floorplanning, we have designed FLINT: an interactive hierarchical floorplanner. FLINT uses input from a schematic diagram that starts at the block function level, and may extend as deeply as the transistor level, to create a circuit hierarchy and an initial layout. The designer uses the graphical output system and the mouse-driven input system of FLINT to explore as many alternate module floorplans as desired, keeping those which he feels might be part of a good solution. The floorplan utility chooses, from among the alternative layouts, the one which best fits into the overall floorplan. When a module occurs in the schematic in more than one location, the floorplans created for the first module instance are available for use in all instances. FLINT chooses the better alternative for the final floorplan in each instance.

To test FLINT, we created a schematic of the AM2901, a common bit slice processor. This part has several differing functional blocks, some of which are used many times in the design. The AM2901 typifies the kind of circuit we designed FLINT to be used with.

1.6 Thesis Outline

This thesis will describe the development of a floorplanning utility. Chapter 2 presents background information on automated circuit design. The emphasis here is on previous work which has taken place, from the early placement algorithms to automated floorplanners. Chapter 3 provides information on desirable features of a floorplanner, and some of the trade-offs which we made during the design. This bulk of this chapter discusses the features and operations of FLINT. Chapter 4 describes how FLINT carries out these operations, the data structures used, and software tools used during development. Chapter 5 presents the results of our efforts, and describes the performance of FLINT in designing two test circuits. This chapter also discusses some of FLINT's shortcomings. Chapter 6 presents our conclusions about FLINT and some impressions about the merits of some of the design trade-offs. This chapter also discusses some possible future development.

Chapter Two: Previous Work

To fully understand where FLINT fits into the scheme of things, one must first become acquainted with some of the procedures which have been used for layout in the past. This chapter begins by presenting a description of the two most common methods used to layout circuits: macro-cell placement and floorplanning with module generation. Algorithms for both of these layout methods have been presented in the literature in the past few years and a survey of this work makes up the bulk of this chapter. The survey starts by presenting some of the standard cell placement algorithms on which many of the floorplanning algorithms are based, although this thesis is concerned with floorplanning, not standard cell placement. After the standard cell algorithms have been discussed, two macro-cell placement algorithms are presented followed by a survey of algorithms which represent some of the more popular and successful floorplanning techniques. Finally this chapter concludes with a discussion on what is missing in current approaches and why our approach can improve on them.

2.1 Circuit Layout Methods

Current automated IC layout techniques fall into three broad categories. The first two are macro-cell placement and floorplanning with module generation. The third category, outside the scope of this thesis, deals with layout at the gate level. Algorithms in this third category perform standard cell placement, PLA generation and gate array layout, as well as several other layout styles. Layout styles such as these are typically used to place small circuits and may be used to generate modules after floorplanning is completed. For the layout of larger circuits, either macro-cell placement or floorplanning can be used.

2.1.1 Macro-Cell Placement

Of the two popular layout methods, macro-cell placement was dealt with first in the literature. It preceded floorplanning historically because it is more closely related to the standard cell placement techniques which, in turn, preceded it. Macro-cell placement treats modules as entities whose shape is fixed and whose terminal locations are known in advance. Although macro-cell placement algorithms are supposed to place and orient modules so that area and interconnect lengths are minimized, these goals often conflict. While modules which have a large number of connections in common should be placed closely together to reduce interconnect length, this may create unused area or dead space in the layout. It may be worthwhile to rearrange the layout slightly to reduce the dead space, even though this may push tightly connected modules farther apart.

In addition to determining the location of modules on the IC, macro-cell placement algorithms must also decide which way cells should be oriented. Changing cell orientations and slightly rearranging the layout can have dramatic effects on the quality of the layout, by reducing both interconnect length and dead space.

The modules used as macro-cells are themselves laid out prior to the macro-cell placement step. This fixes external connections on the macro-cells and suggests another way to improve a layout, namely mirroring a module horizontally or vertically. Mirroring can reduce interconnect length by positioning external connections closer to the other modules to which they are connected. This step is often delayed until after the relative positioning and orientation of the modules.

2.1.2 Floorplanning with Module Generation

Floorplanning, followed by module generation, is an alternative to macro-cell placement. Like macro-cell placement, floorplanning is responsible for placing modules relative to each other. The major difference is that the floorplanner treats some, or perhaps all, of the modules as flexible blocks. On the surface it appears that working with flexible blocks, as opposed to fixed blocks, simplifies the problem, since the shape of the modules can be changed to remove any unused space. But flexible modules can be more difficult to work with, because their area may not remain constant as their aspect ratio is varied. The **shape function**, which describes how the area changes with aspect ratio, may be very complex.

Like a macro-cell placement algorithm, a floorplanner is responsible for allocating area on an IC for each module. In addition, however, it must also determine the shape of the modules. To further complicate the problem, the shape and size of the modules used in floorplanning may just be estimates, since the modules may not be generated until after the floorplanning step. If modules are generated after floorplanning, the module generators must try to produce modules which fit the area and aspect ratio estimates, provided to it by the floorplanner, as closely as possible. Having accurate estimates is important because if the modules generated are much larger or smaller than expected, the layout may waste space. Generating modules after the floorplan is created has some advantages: the global layout information provided by the floorplan can be used by the module generators to improve the layout by optimizing the location of external connections to reduce overall interconnect lengths.

There are trade-offs to be examined in choosing whether floorplanning or macro-cell placement should be used. One advantage to macro-cell placement is that since modules are generated prior to placement, modules which are encountered often can be optimally laid out once and stored in a library for future reference. Another advantage to macro-cell placement is that, since it does not have to optimize the shapes of the modules, it's less complex than floorplanning, making it less time consuming. Of the two approaches, the advantage in terms of potential layout quality must lie with floorplanning. Since a floorplanner can vary module shapes so that they fit together very tightly, and can provide information to module generators on module locations to allow them to optimize terminal location, it can reduce interconnect length. Overall it would seem that spending more computer time with floorplanning is worth while, so long as the increase in computer time is not excessive. The increased flexibility also makes floorplanning the more attractive alternative. It was for these reasons that we decided that a new floorplanner would be created rather than a macro-cell placement utility.

2.2 Literature Survey

2.2.1 Introduction to Layout Organizations

Before beginning the literature survey, something should be said about layout organizations. Layout organizations can be grouped into two categories: slicing structures and non-slicing structures. A considerable number of layout methods have been based upon slicing structures. When partitioned this way, the circuit is viewed as being composed of two partitions which are strongly connected internally, and weakly connected to each other. They are separated horizontally or vertically by a slice line. Each partition is made up of two more partitions which, in turn, are made up of two partitions and so on until each partition consists of a single cell. An example of a slicing structure is shown in Figure 2.1a. Slicing structures can be represented as a binary tree where each internal node represents a slice line, and each leaf node represents a cell. A slicing organization considerably simplifies layout complexity. While slicing does not represent all possible circuit organizations, it represents a great many.

Non-slicing organizations obviously include all the layout organizations which are not slicing structures, but, in the context used here, they may also include some layouts which are slicing structures. General non-slicing structures cannot be represented by a binary tree and that is where the distinction is made. A simple example of a non-slicing structure is the so-called pinwheel layout shown in Figure 2.1b. Note that no slice can partition the layout into two parts. Another non-slicing layout is given in Figure 2.1c. Algorithms designed to deal with slicing structures are less complex and much faster than those which deal with more general organizations, but still produce layouts which rival non-slicing layouts in terms of minimizing area consumption. Some algorithms allow layouts which combine pinwheels with slicing structures to be represented in a binary tree. An example is given in Figure 2.1d.

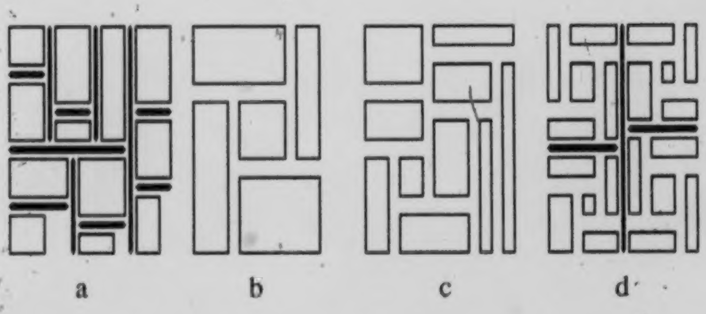


Figure 2.1: Slicing vs. Non-Slicing Structures

At this point it may be prudent to define some terms which will be used throughout this thesis. Block will often be used as a synonym for partition, although block will be used as a noun only and partition may be used as a verb. A module is a block, but also is defined by a list of components which make it up. Components are the items which make up modules. Components may be either cells or module instances. Module instances are place holders within modules, whereas cells are atomic, and cannot be broken down further.

2.2.2 Standard Cell Placement

Since many of the macro-cell placement and floorplanning algorithms are based on standard cell placement algorithms, this survey begins with a look at standard cell placement algorithms. The first algorithms examined are those which use slicing structures and some kind of partitioning.

2.2.2.1 Standard Cell Placement Using Partitioning and Slicing Structures

The partitioning of a circuit can proceed either bottom-up or top-down. Bottom-up partitioning starts by grouping cells together, then grouping the groups together and so on, until the entire circuit has been combined into one group. Binary top-down partitioning starts by separating the circuit into two groups of cells, then partitioning each group into two separate groups, and so on, until each group consists of a single cell.

Bottom-up partitioning, or clustering, as presented by Schuler and Ulrich [ScUl72] starts by examining the connections between all pairs of cells. For each pair, a connection strength value is derived using the number of connections they have to each other relative to the number of connections they have in total. Clustering starts by pairing the two most strongly connected cells. This cluster gets treated as a single cell, and clustering starts over. As clustering proceeds, a penalty for cluster size is included so that large clusters do not just swallow the rest of the cells one by one. A circuit clustered in this way can be represented by a binary tree. The tree can be mapped into a so-called "linear placement" which Schuler and Ulrich [ScUl72] use in a second step. Although their algorithm allows nodes to be rotated (swap the left and right children) to reduce interconnect length, the resulting layouts are usually less than ideal, since the placement will ultimately be two dimensional, not linear.

Another problem with this method is that signals are treated as point to point loops rather than as nets. For example, a single net connecting three cells ***A***, ***B***, and ***C*** needs to connect ***A*** to ***B*** and ***B*** to ***C***. But this method assumes the net connects ***A*** to ***B***, ***B*** to ***C***, and ***C*** to ***A***, biasing the connection strengths unrealistically.

A top-down partitioning method, called min-cut, was proposed by Breuer [Breu77]. Several derivatives now exist, but in general, all the methods separate the cells of the circuit into two blocks of roughly comparable size, let's say blocks ***A*** and ***B***. The blocks are assumed to be separated by a vertical or horizontal cut line. Each cell is connected to a number of signal nets, some attached to cells in ***A***, some to cells in ***B***, and some externally. The objective is to minimize the number of nets crossing the cut line by moving cells back and forth between ***A*** and ***B*** (hence the name min-cut). After an acceptable partition is reached blocks ***A*** and ***B*** are, in the same manner, partitioned into smaller blocks, and the process repeated until each block contains a single cell. A sample partitioning is shown in Figure 2.2. The entire circuit shown in Figure 2.2a has been partitioned into partitions ***A*** and ***B*** in Figure 2.2b. In Figure 2.2c, the cells in partition ***A*** have been partitioned into A_1 and A_2. Figure 2.2d shows the cells in partition ***B*** partitioned into B_1 and B_2. Finally, Figure 2.2e shows the cells of partition A_1 partitioned into A_{11} and A_{12}.

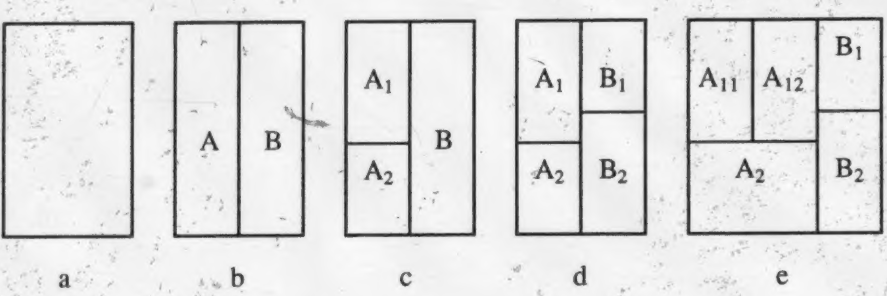


Figure 2.2: Block Partitioning

Breuer based his cell movement on an algorithm of complexity $O(n^2 \log n)$, where n is the number of cells, derived by Kernighan and Lin [KeLi70] for graph partitioning, and later extended by Schweikert and Kernighan [ScKe72].

A faster cell movement algorithm (which does not perform an identical task) was provided by Fiduccia and Mattheyses [FiMa82]. This algorithm is based on Schweikert's and Kernighan's algorithm and is of complexity $O(n)$ where n is the number of cells being partitioned. Briefly, the algorithm proceeds as follows: The initial partition of a group of cells is made randomly. Then several passes are made moving cells back and forth across the new partition. On any given pass, once a cell is moved to the other block, it cannot be moved back. As well, a balance criteria is established so that all the cells do not wind up in the same block. As the cells are moved, the number of nets crossing the cut, known as the cut state, is recorded. After all the cells have been moved once, the best cut state which was encountered during that pass is restored. This new partition is used as the start for a new pass. Passes are performed until no further improvements can be made.

Several contributions have improved the quality of placements produced using the min-cut method. Dunlop and Kernighan [DuKe85] added an extension to partitioning, which they call "terminal propagation". The intent of terminal propagation is to bias the partitioning by considering external connections in order to improve the final placement. If an external net intersects one of the two partitions, say partition ***A***, an immovable dummy cell connected to that net is placed in that partition. This makes it more attractive to move cells connected to that net into partition ***A***. If the external net intersects both partitions, then dummy blocks are placed on both sides of the partition, resulting in it having no affect on the partitioning at all. A key to this method is the way in which points of intersection are determined. Low cost Rectilinear Steiner trees (a net connection scheme) are computed for each net that has some elements inside and outside of the block being partitioned to determine the point of intersection with the block. If this point is close to the cut line, the net is ignored; otherwise, a dummy cell is created on the side of the block where the connection point is located and will thus bias the partitioning. The authors claim that terminal propagation reduces area consumption by up to 30%.

One more step must be added to these slicing placement routines if they are to be applied to standard cells. The cells in the slicing structure must be put into rows. Schuler and Ulrich [ScUl72] create rows by folding their linear placement as many times as needed. Breuer [Breu77] creates rows by starting with horizontal partitioning until enough rows have been generated, and then performing vertical partitioning on each row. Fidducia and Mattheyses [FiMa82] deal only with the partitioning problem.

2.2.2.2 Standard Cell Placement Using Non-Slicing Structures

Although partitioning is a popular method for placement, it is not the only solution. One of the classical techniques for cell placement is force-directed placement. In general, algorithms using this technique have two phases: constructive initial placement and iterative improvement. The initial placement phase selects a seed cell to place in the circuit, then selects unplaced cells one at a time and, by some means, places them in unoccupied locations in the layout. The iterative improvement phase identifies poorly placed cells and moves them to a better location.

One of the later refinements, introduced by Goto [Goto81], focuses primarily on the iterative improvement step. Goto's initial placement step starts by selecting one seed cell and placing it. Unplaced cells are placed around this seed, based on the connectivity which they have to cells already placed. One by one, Goto's algorithm selects an unplaced cell and places it in an empty slot in such a way as to minimize routing length. Goto ranks the locations in which each new cell can possibly be placed and then randomly chooses one of the better ones. The entire procedure is executed from scratch several times to create several different initial placements. After initial placement, Goto's iterative improvement step is applied to each initial placement and the best resulting layout is selected as the final layout. A generalized procedure for this is presented in procedure "replace" given below. It requires a multi-dimensional array ***A***, with one element in the first dimension, ε elements in the second dimension, ε^2 elements in the third, ε^3 in the fourth and so on for λ dimensions (λ is selected before the procedure is run). In the following procedure, ***i_a*** is an index array for the array ***A***.

```
i = 1;            /* i is global */
    Procedure replace(A)
    /* Cell A is poorly placed */
1.  Find ε locations where cell A would be better placed.
2.  For each of these locations do the following (for (j=1; j≤ε))
3.  Put A in location j and displace the cell formerly at j (call it
                the displaced cell B)
4.  If we put B in the location where our original A came from,
                will the placement be better than the
                best one encountered so far (will it
                minimize wire length)? If so save as
                state X.
4.  i = i + 1;
5.  If ( i < λ) then replace (B);
6.  else stop and return
7.  Return state X as the new placement.
```

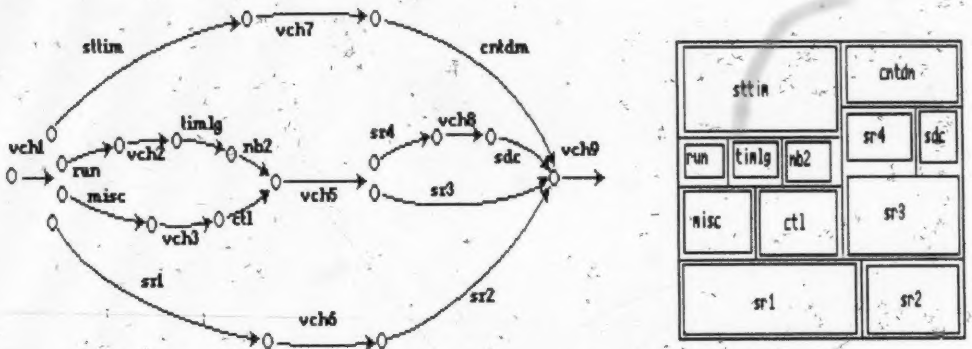
The improvement step consists of selecting a cell *A* which is poorly placed and finding ε positions at which it is better placed (Goto tries various values of ε and finds $\varepsilon = 4$ to be a good value) and proposes exchanging cell *A* with each of the cells at the ε locations. Each cell at the ε locations in turn displaces ε cells, and so one until λ levels of displacement have been tried. This boils down to an exchange of up to λ cells. The exchange which reduces the wiring length the most from all the possibilities is chosen as the one to implement. This iterative improvement technique is a generalization of many of the techniques which preceded it in the literature.

2.2.3 Macro-Cell Placement

Macro-cell placement is conceptually very similar to standard cell placement. There are some important constraints inherent in standard cell placement which one would like to remove. The equal height of standard cells means that cells always fit together in rows and no cell rotations ever need to be considered. However, macro-cells (which have various different heights) often do not fit neatly into rows. These new constraints were first addressed by algorithms designed to perform macro-cell placement. The major additional dimension of control provided by macro-cell placement is over module orientation. Orientation can be modified by mirroring a module horizontally, vertically, by tipping it on its side, or any combination of these. In standard cell placement, cells can typically be stretched vertically but cannot be tipped on sideways or mirrored horizontally.

The two macro-cell placement algorithms presented below use placement techniques which are substantially different from the standard cell techniques discussed earlier. In the standard cell examples, the cells were ultimately placed in rows. If slicing structures are employed in placement and some rows are too long (due to feedthroughs or other problems), it is a simple matter to promote some cell, or partition, to an adjacent row. In macro-cell placement, this is not possible; this limitation reduces the appeal of using slicing structures. Neither of the macro-cell algorithms discussed here uses slicing structures.

Preas and vanCleemput [PrVa80] present a graph based placement model. A pair of graphs, called channel intersection graphs, are used to represent the horizontal and vertical relationships between blocks and routing channels. An example of a horizontal channel intersection graph for the layout of Figure 2.3b is given in Figure 2.3a. On this graph, vertices represent the boundary between channels and blocks, while edges represent the width of a block or a channel. The graphs, which start out representing partial placements, are used to create layouts by adding blocks to the graph one at a time.


sltim
vch7
cnkdm
vch1
timlg
vch8
vch9
run
vch2
nb2
sr4
sdc
misc
vch5
sr3
vch3
ctl
sr1
vch6
sr2
sttim
cntdn
sr4
sdc
run
timlg
nb2
misc
ctl
sr3
sr1
sr2

a b

Figure 2.3: Channel Graph and Floorplan

If the number of blocks to be placed is small then the following exhaustive procedure is used.

```
RECURSIVE PROCEDURE placeblock;
FOR each possible position on channel intersection graph DO
    BEGIN
    derive new graphs for every unplaced block at this
                        position;
    FOR each graph DO
        BEGIN
        FOR each allowed orientation for this block DO
```

```
                    BEGIN
                    place this block in this position and orientation;
                    IF resulting area is smaller than present bound
                            and size constraints are satisfied
                            THEN
                        BEGIN
                        IF there are currently unplaced blocks
                            THEN
                          placeblock;
                        ELSE IF aspect ratio constraints are
                            satisfied THEN
                          save this placement as new bound;
                        END;
                    END;
                END;
            END;
        END;[1]
```

This procedure basically tries every block in every location using every orientation, discarding those partial solutions which are already larger than the smallest one found so far. This is very time consuming so, when a large number of blocks need to be placed, an initial placement is created by selecting some seed blocks, making a graph for these, and then placing the rest of the blocks. The procedure for this follows.

```
        PROCEDURE initial_placement;
```

---

[1] Preas and vanCleemput "Placement Algorithms for Arbitrarily Shaped Blocks"

```
IF seed is specified by user THEN
    read user_specified seed;
ELSE
    construct a seed;
derive graphs for seed;
WHILE any unplaced blocks exist DO
    BEGIN
    set bounds on maximum height and width permitted;
    modplaceblock (1, max_recursion_depth);
    get best result to date, make that current;
    END;[2]
```

The location of new blocks in the graph is not determined exhaustively when the number to be placed is large. Rather, blocks are selected for addition in some order, and after trying a number of combinations of additions, the first block of the combination which results in the smallest partial placement is added to the graph. This procedure follows:

```
RECURSIVE PROCEDURE modplaceblock(level, maxlevel)
FOR each possible position on channel intersection graph DO
    BEGIN
    derive new graphs for a block in this position;
    FOR a subset of unplaced blocks DO
        BEGIN
        place this block in this position;
```

---

[2] Preas and vanCleemput "Placement Algorithms for Arbitrarily Shaped Blocks"

```
            IF detailed size is smaller than current bound and
                        constraints are satisfied THEN
              BEGIN
              IF all blocks are placed OR level >= maxlevel
                        THEN
                  save this placement as new bound;
              ELSE
                  modplaceblock (level+1, maxlevel);
              END;
            END;
        END;
    END;2
```

Two factors limit the number of combinations tried. First, only a subset of blocks is selected to be considered for addition to a given graph, and then the number of blocks actually added is limited (to say three blocks). The subset of blocks chosen to be considered is based on block geometry, and connectivity. Since this procedure is unlikely to result in a really good placement, the authors also have an iterative improvement step where they uproot some of the blocks and try and replace them. The details of block selection for uprooting and replacement were presented by Preas [Prea79]. The authors compare the placement of a circuit done with a standard cell placement algorithm described by Feller [Fell76] and their procedure. They first partitioned the circuit and estimated block dimensions (by hand?), used their procedure to come up with a trial placement, laid out the blocks using the algorithm given by Feller, obtaining exact block dimensions. Then they re-ran the macro-cell placement routine, finally ending up with a circuit size about 3/4 the size of that produced by

Feller's algorithm. While this serves to point out the deficiencies of standard cell placement procedures when applied directly to large circuits, it also points out some problems with the authors procedure, namely that some manual intervention must be taken by the designer to partition the circuit and to estimate block dimensions. The circuit they tested their procedure on was relatively small, and it is unknown how much time was taken or how well large circuits would be placed.

Some methods of solving macro-cell placement problems are rather unconventional. One method, discussed by Sha and Dutton [ShDu85], requires a new definition of a rectangle: one where the ends are actually semicircles, so that they may reformulate the problem as a non-linear optimization problem. In order to solve a problem in this manner, the problem must be stated as an objective function, subject to a number of constraints. The objective function used by the authors is minimization of wire length. The constraints are on block orientation, block shape, block overlap, and chip aspect ratio.

The authors model rectangles as a line segment and a radius, so that the ends of the rectangles are actually semi-circles. Normally a rectangle is defined by the coordinates of a point, a height, a width, and an orientation (either 0 or 1). The new model defines rectangles using two endpoints (x_1, y_1), (x_2, y_2) and a radius r. Orientation is defined either as $(x_2 - x_1)/l$, or as $(y_2 - y_1)/l$, where l is the length of the rectangle from endpoint to endpoint. The advantage of this model is that one formula describes rectangle overlap in all directions, thereby reducing the number of equations. The trade-off is that rectangles may overlap after placement because the description is inaccurate.

Net length is approximated by finding the gravitational center of all blocks attached to the net, and summing the distance from this centerpoint to each block attached to the net. If, for block B_i, its center is at (x_i,y_i), and there are m blocks attached to net k, the center of net k is at (X_k,Y_k), where

$$X_k = (1/m)*\sum_{i=1}^{m} x_i \quad \text{and} \quad Y_k = (1/m)*\sum_{i=1}^{m} y_i.$$

The modified wire length w_k for net k is given by

$$w_k = \sum_{i=1}^{m} (x_i - X_k)^2 + (y_i - Y_k)^2.$$

The objective function, which is to minimized is given by

$$f(x) = W = \sum_{k=1}^{n} w_k.$$

The restraints (whose formulae are not given in this thesis) are:

$g_1(i, j) \le 0$	block overlap
$g_2(i) = 0$	block orientation
$g_3(i) = 0$	block shape (derived from rectangular width and height)
$g_{4kl}(i) \le 0$	perimeter constraints due to aspect ratio

for $l = 1\text{-}2$; $k = 1\text{-}4$; i and j=1-m, $i \neq j$,

where n is the number of signal nets in the circuit. This formula, and those used to describe the constraints, are non-linear. The steps involved in solving the problem are:

1) Select a series c_k which progresses to infinity, such as $c_0 = 1$, $c_{k+1} = 10c_k$.

2) Construct a new unconstrained objective function:

$$P(x, c_k) = f(x) + c_k \sum_{j=1}^{4} [Max(0, g_j(i))]^2$$

3) Use and unconstrained optimization technique to minimize $P(x, c_k)$, iterating for k until the problem is solved.

The authors use Newton's method to solve the problem.

The authors also relate ***W*** to the attractive potential energy of a gravitational system, but it is unknown why this number should be preferred to unsquared wire length. Nonetheless, the example placement provided by the authors, which has 15 blocks and 142 nets, compares favourably with a manual placement of the same circuit. The area is 3% larger than the manually placed circuit and the wiring length is 13% shorter. This is an excellent result, but more comparisons are needed to draw any substantial conclusions.

2.2.4 **Floorplanning**

The inability of macro-cell placement algorithms to manipulate module shapes is a great drawback, which results in unused area or dead space. Floorplanning removes this restriction by allowing the determination of module shape to be an integral part of the placement process. While this gives floorplanning greater flexibility, it also increases the complexity of the layout problem. Although automated floorplanning is a recent topic of research, it has already received substantial attention in the literature.

2.2.4.1 Floorplanning Using Slicing Structures

Before presenting an algorithm to determine optimal cell orientation in slicing floorplans, we must first define the cell orientation problem. Assume we have a floorplan (such as that of Figure 2.3b), which is defined by a series of intersecting lines (lines may not cross), which divide the floorplan into rectangles. The outermost perimeter must also be a rectangle which we will call the boundary rectangle. Each rectangle not crossed by a line is called an inner rectangle and each inner rectangle encloses a module. The enclosed module must not have dimensions greater than those of the enclosing rectangle. Some freedom is allowed in the dimensions of the inner rectangles which make up the floorplan, but the relationships between the inner rectangles may not change. For example, in Figure 2.3b, the rectangle **timlg** is to the right of rectangle **run**, and to the left of rectangle **nb2**. It is above **sr1** and below **sttim**. Although the dimensions of the rectangles may change, these relationships must not, or the floorplan has been altered. The relationships between inner rectangles can be captured with a pair of graphs, one defining vertical relationships, and the other defining horizontal relationships (similar to Figure 2.3a, but without the channel segments). Two floorplans with the same graphs, but different rectangle dimensions are equivalent. If the dimensions for a module are given by (a, b), then the module can have two orientations, (a, b) and (b, a). The optimal orientation problem is to find a set of module orientations for each module, such that the total area of the boundary rectangle is minimized.

The orientation of cells within a floorplan may seem to be more readily applicable to macro-cell placement, but this problem has actually received more attention in the field of floorplanning.

An algorithm for determining optimal module orientation in slicing structures was presented by Stockmeyer [Stoc83]. He showed that for the general non-slicing floorplan, optimal module orientation is NP-complete; however, for slicing floorplans an algorithm, which is no worse than $O(n^2)$, (where n is the number of modules) and is typically $O(n \log n)$, is presented. His algorithm combines, in linear time, a set of boundaries for each of two modules, giving a set of boundaries for the pair. This procedure combines the boundaries of pairs of nodes in a bottom-up manner. A leaf node with dimensions a and b, $a > b$ starts with the set of boundaries $\{(a,b),(b,a)\}$. In general, a non-leaf node would have as its set of boundaries $\{(h_1,w_1) \ldots (h_k,w_k)\}$, where $h_i > h_{i+1}$ and $w_i < w_{i+1}$ and k is the number of boundaries. If we have two nodes U and V whose boundary sets are $\{(h^U{}_1,w^U{}_1) \ldots (h^U{}_k,w^U{}_k)\}$ and $\{(h^V{}_1,w^V{}_1) \ldots (h^V{}_m,w^V{}_m)\}$ and they are to be combined, the generalized procedure is:

```
Procedure Combine (($h^U$,$w^U$), ($h^V$,$w^V$), slice_dir)
/* ($h^U$,$w^U$) and ($h^V$,$w^V$) are arrays of boundary pairs */
BEGIN
Initialize $i$ =1, $j$ = 1;
WHILE $i < k$ and $j < n$ DO
    BEGIN
    add Join(($h^U_i$,$w^U_i$), ($h^V_j$,$w^V_j$), slice_dir) to the list with
                                pointers to ($h^U_i$,$w^U_i$) and ($h^V_j$,$w^V_j$);
    IF $h^U_i > h^V_j$ THEN increment $i$;
```

```
        IF $h^u_i < h^v_j$ THEN increment $j$;
        IF $h^u_i = h^v_j$ THEN increment $i$ and $j$;
        END;
    END;

    Procedure Join(($h^u_i$,$w^u_i$), ($h^v_j$,$w^v_j$), slice_dir)
    BEGIN
    IF slice_dir is horizontal THEN $(x,y) = (h^u_i+h^v_j, \max(w^u_i,w^v_j))$;
    ELSE $(x,y) = (\max(h^u_i+h^v_j), w^u_i+w^v_j)$;
    return $(x,y)$;
    END Procedure;[3]
```

For k elements in the U set and m elements in the V set, this combining operation requires at most $O(k + m)$ steps. Since the leaf nodes have at most two orientations each, the top node of the tree can have at most $2n$ pairs in its list, where n is the number of elements in the circuit. This algorithm has great importance to floorplan automation because it provides a quick way to estimate the effect of module orientation. This effect is calculated by replacing the two dimensional pairs at the leaf node in question, and then recalculating the area for the entire circuit using the aforementioned algorithm. The area can be recalculated in $O(n \log n)$ time.

Some of the standard-cell placement algorithms discussed so far have been adapted for use in floorplanning. Converting these methods to floorplanning requires adding extra steps to address the additional module shape problem. The new algorithms usually undergo some refinement as well, since the shortcomings of the method become better understood with time and improvements can be made.

---

[3] Stockmeyer "Optimal Orientations of Cells in Slicing Floorplan Designs"

A floorplanner called Mason [LaDi85] uses some variations on the min-cut method of placement in combination with Stockmeyer's algorithm. Mason changes the weight of the nets crossing the cut line as the sizes of the partitions change, increasing the weight as the difference in size increases. This limits the probability of moving modules from the smaller to the larger partition. Mason further enhances performance by exhaustively partitioning small groups of modules (the authors suggest 18 or fewer modules). Mason's performance is further enhanced by the addition of terminal propagation. Terminal propagation in Mason appears to consider only the direction of terminal connections rather than creating Steiner trees, as was done in Dunlop and Kernighan [DuKe85].

To convert the slicing tree to an actual geometric layout, Mason uses an orientation algorithm developed by Otten [Otte83], which in turn is based on Stockmeyer's [Stock83]. This algorithm uses a shape function for each leaf node, modeled by continuous piecewise linear curve segments, and combines shape functions for the children together at each internal node. After choosing a geometric layout with Otten's algorithm, a global routing step estimates routing area, using the slices as routing channels. This updated area information is used to choose a new final geometric layout.

A floorplanner similar to Mason was discussed by Modarres et al. [Moda88]. The major difference between it and Mason is that this floorplanner uses a multi-level circuit hierarchy, whereas Mason used a two-level hierarchy (modules and cells). The hierarchy is determined in advance of the floorplanning and, presumably, corresponds to a schematic of the circuit, also defined hierarchically. Each level in the layout hierarchy will be referred to as a module. Each module in the hierarchy is composed of several module instances, which have yet to be partitioned. The circuit is placed in a slicing tree in which the module instances which make up a module are its children. At this stage there are no slicing nodes between the module nodes and the module instance nodes (since no partitions have been created yet). The tree is traversed several times during floorplanning. The first traversal estimates the area of each leaf module and derives a hyperbolic function of the x and y dimension to represent it. The second traversal partitions the components of each module in the tree using the Kernighan and Lin partitioning algorithm and a simplified version of terminal propagation (this is the step which adds slicing nodes between the module nodes, to give a slicing tree). The partitioning includes a penalty for moving modules from smaller to larger partitions. If there are less than 15 modules in a partition, they are partitioned optimally. The third traversal of the tree calculates the shapes of the internal modules using an algorithm similar to Otten's [Otte83]. The fourth traversal chooses the shapes of the modules and the fifth estimates channel size by using a global routing step. Two more traversals use the routing information to update the module shape functions. These last two steps result in a complete floorplan which can be manipulated manually if desired. The authors provide the CPU times taken to floorplan various circuits, but provide no information on circuit areas or comparisons with other floorplanning methods.

All the floorplanners considered so far examine only a small part of the floorplanning search space and converge on a local minimum which is unlikely to be optimal. Min-cut explores beyond a local minimum, but not very far. A method which will explore farther beyond a local minimum is simulated annealing. This method is based on a model of the physical process of annealing. At the beginning of the process, module placement is random and the circuit is assumed to be at a high temperature. High temperatures represent very disordered states in which modules may move about freely. As the temperature cools, module movement becomes more restricted and eventually the layout settles into a position of low energy or low stress, meaning the modules are well placed. The main feature of simulated annealing is that possible moves are generated randomly and evaluated. If a move is beneficial, (reducing total wire length), then the move is accepted. If the move is not beneficial, then a random number is generated and depending on the random number and the cost of the move, the move may still be accepted. A general simulated annealing algorithm follows.

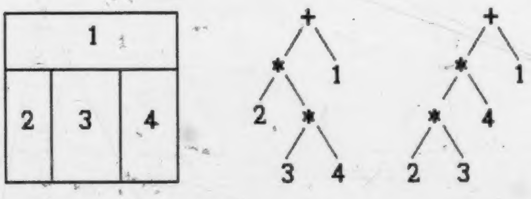
```
Anneal (state$_0$, T$_0$)
/* state is the initial placement state
T$_0$ is the initial temperature */
1.  T = T$_0$;
2.  state = state$_0$;
3.  while (T is higher than T$_{stop}$)
4.      num_states_generated = 0;
5.      while (num_states_generated < CONSTANT)
        /* CONSTANT is based on the number of
                components (n) being placed and the
                type of component being placed. For
                standard cells this may be n * 50. If
                components are macrocells this may
                be n * 15 */
6.          generate new state based on State
                (State$_{proposed}$);
```

```
7.              cost_new = Evaluate state ($State_{proposed}$);
8.              cost_old = Evaluate state (State);
9.              if (cost_new < cost_old)
10.                 State = $State_{proposed}$;
                else
11.                 $\Delta$cost = cost_new - cost_old; /* will be < 0 */
12.                 if ($e^{(\Delta cost/T)}$ > (random number between 0
                            and 1))
13.                     State = $State_{proposed}$;
                    endif
                endif
14.             increment num_states_generated;
            endwhile
15.         lower T by some small amount;
        endwhile
```

To lower the stress, a module must be placed close to the other modules with which it shares connections. We shall refer to modules which share a large number of nets as close neighbors, even if they are not physically close in the layout. When the temperature is relatively high, the probability of moving modules away from their close neighbors, thus increasing layout cost, is high, although the probability of moving them closer together is higher still. As the temperature decreases, the probability of moving close neighbors farther apart decreases, while the probability of moving them closer together, thus decreasing cost, remains high. Proposed new states which decrease the cost are always selected. The distance over which a displacement may occur decreases with temperature.

The performance of simulated annealing is very sensitive to a number of parameters. Changing the temperature, number of proposed new states, distance criteria and the new state generation procedure all play important roles in determining the quality of the final solution and the length of time required to achieve it. While simulated annealing produces some of the better placement results, it also takes the greatest amount of time, sometimes taking many hours to place circuits of modest size.

Simulated annealing is applied to slicing floorplans by Wong and Liu [WoLi86]. The authors develop a way of uniquely describing slicing structures with normalized Polish expressions, thus simplifying the floorplanning process. The Polish expressions represent slicing trees whose internal nodes have a slice direction associated with them and whose leaf nodes represent modules. Not all slicing trees or Polish expressions are unique, and there is no one-to-one correspondence between floorplans and slicing trees. Figure 2.4 shows a floorplan and two possible slicing trees.



a b c

Figure 2.4: Floorplan and Two Slicing Trees

If we force the right child of a slicing tree to have a slice direction different from that of its parent, then the slicing tree is guaranteed to be unique. Even with this restriction all slicing floorplans are possible, and we have the desired one-to-one correspondence between slicing trees and slicing floorplans.

Polish expressions are formed with the two complementary operators '*' and '+', which represent vertical and horizontal slices respectively, and by numbers which represent nodes. Thus the Polish expression 2 3 * 4 * 1 +, represents the tree in Figure 2.4c. For an expression to be a proper Polish expression, every operator must be preceded by a greater number of operands than operators, so proper Polish expressions must start with two operands and must end with an operator. Proper Polish expressions must not have consecutive similar operators since right children in a tree must have a different slice direction than their parents to ensure uniqueness. Examples of invalid sequences are 1 2 * 3 4 + +, * 1 2 +, and 1 2 * + 3.

There are three types of operations which can be performed on the tree: The first interchanges the position of two adjacent modules in the tree by interchanging two adjacent operands in the Polish expression. The second complements a string of operands thereby changing the slice directions of some nodes in the tree. The third operation selects an adjacent operator and operand and exchanges them, which reorganizes the tree. The first two operations obviously leave a proper expression, provided they started with one, but the third operation does not guarantee this, so the expression must be checked for legality after the third operation is performed. Legal examples of these operations are given in Figure 2.5.

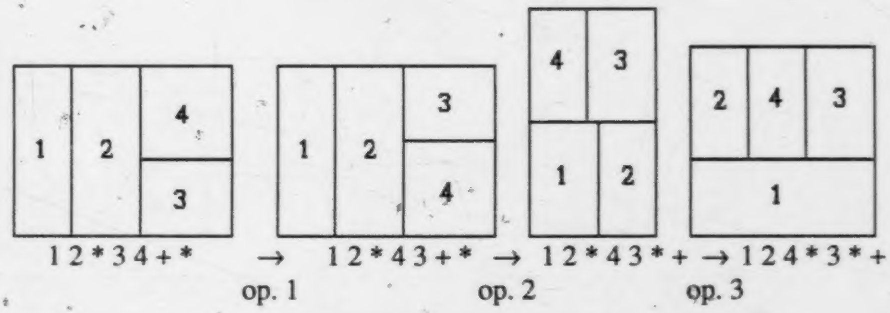


Figure 2.5: Floorplans and Proper Polish Expressions

The cost function of a particular floorplan is based on circuit area and wire length. The wire length is approximated by the Manhattan distance between module centers multiplied by the number of connections between them, summed over all pairs of modules. The area of the circuit is computed using the modified Stockmeyer algorithm presented by Otten [Otte83] for piecewise linear curves. After every modification to the circuit, a new area is calculated, the centers of the modules are determined, and a new wire length is computed. The cost function is of the form $\Psi = A + \lambda_w W$, where A is the area, W is the wire length, and λ_w is the relative weighting of wire length to area and ranges from 0 to 3 with a typical value of 1. The temperature in the annealing schedule is changed by a constant rate, and at each temperature, enough moves are tried until a predetermined number of downhill moves are accepted, or simply twice that many moves are attempted. The process stops when less than 5% of all moves made at any given temperature are accepted. The results indicated for up to 40 modules are good with reasonable restrictions on module aspect ratio, final chip area, and various weightings on wire length vs area in the cost function. However, no comparisons are made to other floorplanning algorithms, and no CPU times are given to indicate the algorithms efficiency.

Another process similar to simulated annealing, called a sequence heuristic, was introduced by Nahar and Sahni [Naha86]. The sequence heuristic accepts only moves which improve the solution. To explain briefly, when the last n moves generated have not improved the solution, then the current floorplan is assumed to be in a local minimum. To escape from this position, m random moves are generated and accepted (regardless of their effect on the placement). Some predetermined number of local minima are investigated. This particular implementation of the sequence heuristic is based on the simulated annealing algorithm used by Wong and Liu. Other aspects of this algorithm were covered in the preceeding section.

2.2.4.2 Floorplanning Using Non-Slicing Structures

The force-directed placement method has been adapted to floorplanning in CHAMP [Ueda85]. This method includes a force-directed initial placement, in which the blocks may overlap, followed by block unpacking and block reshaping steps to remove overlap. Initially a connection matrix $\boldsymbol{E}$ is constructed, where entry $\boldsymbol{E}_{ij}$ is the number of connections between modules i and j. The matrix is used to calculate attractive and repulsive strengths, where strongly connected blocks attract and weakly connected blocks repel. These forces are used to find relative blocks placements. The method starts by randomly choosing a block and, using attractive forces, moving it to its gravitational center. From this new position repulsive forces are used to calculate a newer position for this block and it is moved there. After each block has been moved the process is iterated several times until it converges to a stable state. This ends the initial placement phase, but the layout may still contain considerable block overlap. The second step, block unpacking, begins by assigning a shape to each flexible block and creating a chip boundary. Blocks are unpacked by repeatedly selecting and applying one of three procedures: The first procedure simply reduces the chip boundary by a small value. In the second procedure a module is randomly selected, and temporarily shifted a small amount in all four directions. The shift which reduces block overlap and boundary overlap the most is implemented. In the third procedure, a block is randomly selected and its shape temporarily modified by a small amount. Both shorter/wider and taller/narrower shapes are proposed and again the shape which reduces overlap the most is implemented. When overlap between blocks is eliminated, and a satisfactory size is achieved, the process is terminated.

Blocks in this model are either generated prior to floorplanning or they are made up of standard cells. Module shape is changed by adding or removing single rows of cells. Calculation of block width involves adding some extra width at the end of rows since the cells are unlikely to fit perfectly. Calculation of block height involves using information on channel density, obtained experimentally, and is a function of the total width of all cells in the block, number of rows, number of nets and number of pins per net. Generally good results are claimed but dead space still remains in the layouts and must be removed manually.

2.2.4.3 Comparison of Algorithmic Floorplanning Methods

Some of the approaches to floorplanning algorithms are compared by Cai and Hegge [CaHe88]. The algorithms compared are min-cut, with the method of terminal propagation presented by Dunlop and Kernighan [DuKe85], a force-directed procedure similar to CHAMP [UeKi85], simulated annealing as described by Wong and Liu [WoLi86], and a sequence heuristic as described in [Naha86]. Cai and Hegge run two trials: In the first trial both fixed and flexible blocks are used in the floorplans (about half of each). In the second trial only flexible blocks were used. Their results indicate min-cut outperforms the force-directed method used in CHAMP and also outperforms simulated annealing and sequencing, but only if the CPU time allotted is reasonably limited. Sequencing produces better results than simulated annealing, if the CPU time is limited. Simulated annealing and sequencing produce better IC aspect ratios than min-cut or the force-directed methods, and better areas if allowed to run for long periods of time. Both simulated annealing and sequencing can be used to improve the results of min-cut and force-directed floorplanners in terms of aspect ratio but, interestingly, for the largest circuit tested, this resulted in a substantial degradation of size and wiring length relative to min-cut alone, even when they were allowed to run for approximately 1½ hours. The authors note that the parameters used for simulated annealing and sequencing can have a marked effect on their performance, and that more experimenting would be needed to optimize the parameters.

2.2.4.4 Expert System Floorplanning Methods

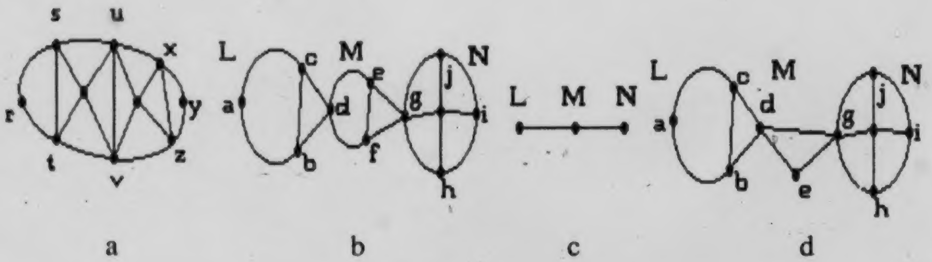
While algorithmic approaches to circuit layout take considerably less time to complete than manual designs, they are still unable to equal human layouts. One proposed solution is to use expert systems, or knowledge-based approaches for layouts. In general, these approaches examine the layout after each design step and decide what step to take next based on a set of heuristic rules. The system tries to look for those features which a human expert would look for, evaluate their relative importance, and make a decision the same way a human would. Quantifying everything a human expert knows about layout would be a formidable task. Thus, in practice, expert systems only roughly approximate the reasoning of a human expert. Nonetheless, some of these systems show promising results.

Flute [WaAc87] is a two stage floorplanning expert system. The first layout stage, topological layout, places modules on a grid. The modules on the grid are connected by arcs representing module interconnections and off chip connections. Crossovers in the arcs are replaced by routing modules. Modules are placed on the grid in an order determined using expert rules. Those rules which determine the order of selection heavily favor modules attached to high priority buses, and place all the modules on one bus before moving on to the next. Those rules which determine module locations try to place a given module adjacent to other modules to which it is connected. When a new module is placed which causes a crossover in the routing arcs, expert rules are invoked to try and reroute the offending net through existing routing modules. If rerouting does not work, then a new routing module is added to the graph and the interconnections are rerouted using the new routing module.

Flute's second layout stage, geometric realization, provides the modules with shapes and sizes and places them in their final location. Geometric realization is accomplished by treating the problem as a minimization problem and then solving it heuristically. The arcs of the graph are assigned non-unique variables which will be used to formulate constraints. One constraint might be that modules ***A*** and ***B*** should have a border long enough to accommodate three connections between them. If the area of module B is at least 1000 units, another constraint might be width*height ≥ 1000. In a reasonable time there is no known way to optimally solve the minimization problem the authors derive, so Flute uses a heuristic technique similar to that used in CHAMP [UeKi85]. Basically, Flute constructs one graph describing east-west module dimensions, and another describing north-south module dimensions. A module is found which is on the longest path of one graph, but not the other, and its dimensions are changed by a small amount, keeping the area the same. This process is repeated until no module can be found which is on the longest path of only one graph. The authors were encouraged by the results they had achieved at the time of writing, but had more rules which they wanted to try.

Another graph-based method of floorplanning converts planar graphs of layouts to rectangular duals. A rectangular dual is a graph whose nodes represent modules and whose edges represent module adjacencies. The planar graph is derived from the connectivity graph of the circuit, whose nodes represent modules and whose edges represent a connection between two modules. A properly formed graph of this type can be converted to a layout with relative ease. An important step in this method presented by Kozminski and Kinnen [KoKi84], provides a set of criteria to determine if a triangulated planar graph admits a rectangular dual, along with a linear algorithm to determine this. The first criteria for admissibility is that the graph must be triangulated, which means that all faces in the graph are triangular. The second criteria is that all cycles which are not faces must have at least four edges, or arcs. The third criteria is that the graph must be planar, meaning arcs do not crossover each other. The final criteria is that vertices which lie on the outermost cycle have at least three edges attached to them, while all internal vertices have at least four edges attached to them.

Kozminski and Kinnen also provided an $O(n^2)$ (where n is the number of nodes on the graph) algorithm to convert a graph which meets these criteria into a rectangular dual. Before the algorithm is explained, some terms must be defined. The first terms to be defined are the shortcut and the corner implying path. Assume that the graph to be converted to a rectangular dual is called ***G*** and refer to Figure 2.6a. A shortcut on ***G*** exists between nodes ***i*** and ***m*** if nodes ***i*** and ***m*** are connected by an edge which is not part of the outside path of ***G*** .



a b c d

Figure 2.6: Planar Triangulated Graphs

If the path ***(i, j, ... l, m)*** is on the outside path then ***(i, j, ... l, m)*** is said to be a corner implying path, but only if no other shortcuts exist in the path ***(j, ... l)***. In Figure 2.6a, ***(s, t)***, ***(u, v)***, and ***(x, z)*** are shortcuts, but only ***(s, t)*** and ***(x, z)*** are corner implying paths. A corner implying path is said to be a critical shortcut. For the special case where a graph is a triangle, any pair of edges is a corner implying path, even though there are no shortcuts. Another term which needs to be explained is the cut vertex. A cut vertex is one which, if removed, would separate the graph into two or more distinct sub-graphs called blocks. In Figure 2.6b both ***d*** and ***g*** are cut vertices and ***L***, ***M***, and ***N*** are blocks where ***L*** would consist of vertices ***(a, b, c, d)***, ***M*** would consist of ***(d, e, f, g)*** and ***N*** would consist of ***(g, h, i, j)***. The next term that must be explained is the block neighborhood graph. If each block is treated as a single vertex, and edges are added between blocks to indicate a vertex originally shared by the two blocks, then this is called a block neighborhood graph (BNG). The BNG for the graph in Figure 2.6b is shown in Figure 2.6c. Finally a critical corner implying path in one of the blocks of a plane graph is a corner implying path which does not contain any cut vertices in its interior. Note that since block ***M*** of has a cut vertex in its interior, this is not a critical corner implying path. In Figure 2.6d, block ***M***, consisting of vertices ***(d, e, g)***does have a critical corner implying path.

For a graph to have a rectangular dual, its BNG, if one exists, must be a path as in Figure 2.6c, not a cycle, and the end blocks of the path must have no more than two critical corner implying paths each. No other block in the path must have a critical corner implying path. By this definition the graphs of Figures 2.6a and 2.6b have floorplans while the graph of Figure 2.6d does not. Possible floorplans for the graphs of Figure 2.6a and 2.6b are given in Figures 2.7a and 2.7b.

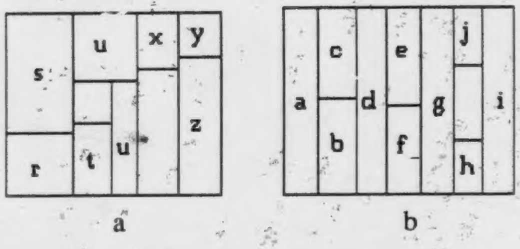


Figure 2.7: Rectangular Duals

Kozminski's and Kinnen's algorithm for converting graphs to rectangular duals begins by creating an outermost cycle for the graph which has four vertices and is called four-completion. The new vertices are connected together in a cycle, and each of the new vertices is connected to at least one interior vertex. An example of a four-completion for the graph of Figure 2.6a is shown in Figure 2.8. The new vertices correspond to the top, bottom, left and right edges of the floorplan. Adjacent vertices on this outside cycle must have one interior vertex (from the original graph) in common. An interior vertex connected to at least two vertices on the outside cycle is called a corner vertex. In Figure 2.8, vertices *s*, *r*, *y*, and *z* are corner vertices. The corner vertex represents a module which will occupy a corner of the floorplan. If the graph has corner implying paths, each path must have at least one corner vertex. This means that there may be no more than four corner implying paths in the graph.

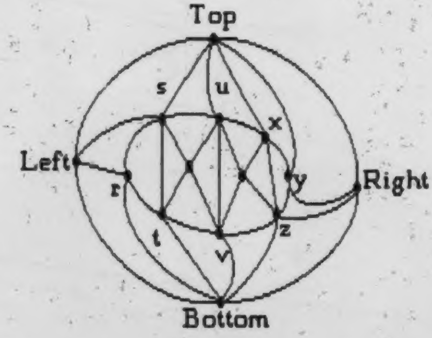


Figure 2.8: Four-Completion of Planar Triangulated Graph

The paths (*s*, *r*, *t*) and (*x*, *y*, *z*) are corner implying paths. Any vertex on the interior of the path may be chosen, and the algorithm provides no guidelines on selecting one over another. Note that because of this restriction, both *r* and *y* **must** become corner vertices. It is not sufficient to choose an endpoint of the path.

Once the graph is converted to a four-completion, and meets all the conversion criteria, the only thing which keeps it from defining a floorplan is adding in edge orientation. Only a brief description of this process will be given here as complete details of adding edge orientations are available in the authors paper [KoKi84]. Adding an edge orientation to the graph involves recursively finding the shortest interior path from the top to the bottom of the graph. This path is used to separate the graph into two parts. A new left node is added to the right part of the graph and a new right node is added to the left part of the graph. Each edge along the path separating the two parts is assigned a horizontal direction in the dual graph and the process repeated for each subgraph. Eventually the subgraphs become small enough to be handled as one of several cases and can be defined completely.

There are a couple of issues associated with creating rectangular duals which are beyond the scope of this algorithm. First of all, the dual of a graph created by this procedure is not unique, and may not be the best possible dual for the given triangulated graph. Secondly, conversion of the original connectivity graph into a triangulated graph requires removing crossovers, meaning edges from the graph must be removed. Removing edges, in effect, removes module adjacencies and decisions must be made as to which edges will be removed. These considerations are addressed by Jabri and Skellern [JaSk87] who employ a knowledge based system to make these decisions.

Jabri's and Skellern's [JaSk87] floorplanner starts by building a graph of the circuit where vertices represent modules and edges represent communication paths. This graph is adjusted to become a graph which admits rectangular duals using knowledge of the modules, interconnections, and the design tools which will be used to build the modules. First impossible adjacencies are deleted (those which cause the graph to violate the criteria presented by Kozminski and Kinnen [KoKi84]) and the number of edges in the graph reduced. From this graph the entire family of rectangular duals are found. This family of duals is reduced by a selection process which first assigns figures of merit to the duals. The figure of merit is based on architectural and design tool constraints, using rules which describe architectural preference based on function. Functional descriptions and the number of interconnections are used to find minimum common border lengths for each possible solution. Using these border lengths, optimal block shapes and minimum layout perimeter are determined with a Simplex algorithm. The perimeter is in turn used to find the circuit and module areas. The floorplanner has been used successfully to design circuits of up to 30,000 transistors. The authors point out that, in one instance, a twelve module graph produced a family of 54 rectangular duals. This is quite a number of potential floorplans to select from, given that all module adjacencies have been determined prior to this. The authors suggest that some form of clustering could be used prior to starting when a large number of modules are present.

2.3 Inadequacies in Floorplanners

The vast majority of floorplanners make minimal use of the hierarchy available in the circuit schematic. They view the circuit as being composed of modules, which are composed of gates, rather than as a few modules composed of a few modules, etc.

Those which do use a hierarchy create it by clustering or min-cut, rather than extracting it from the schematics. This either results in floorplans with many modules, or floorplans with several very large modules. This means that either the floorplanner or the module generator must deal with a large number of items to place. Existing floorplanners also reduce the solution space to a single floorplan, perhaps allowing manual cleanup of the final decision. As a result, the designer has no access to promising intermediate decisions which were rejected by the floorplanning program. Few tools are available which try and make use of a human designer's abilities (not to be confused with expert systems which try and mimic it) while handling the large numbers of recalculations which go along with proposing design changes.

2.4 How Are These Inadequacies Addressed Here?

By creating a circuit hierarchy from the schematic diagram before floorplanning begins, the number of modules which have to be dealt with at any time is greatly reduced. Once the circuit is partitioned along functional boundaries, the solution space is significantly reduced, but the probability that good solutions will be overlooked is still low. By allowing several alternative layouts at any level of the hierarchy, promising layouts of modules can be created and kept under consideration until the final floorplan is chosen, rather than being forced to choose one alternative prematurely. Keeping a human designer in the process increases the likelihood that good solutions will be investigated which an automated process may overlook. FLINT handles all calculations, so the designer can see immediately what effect their changes will have on the final floorplan.

Chapter Three: FLINT: A New Floorplanner

We have developed FLINT in order to address some of the deficiencies of the floorplanners and macro-cell placement algorithms reviewed in the previous chapter. This chapter begins by discussing the merits of various approaches to floorplanning and features which floorplanners might have, and continues by presenting the objectives to be met by FLINT. Next an overview of the system is provided, and finally the details of FLINT's features are presented.

3.1 Definitions

Before continuing it should be pointed out that, in the remainder of this thesis, floorplans are assumed to be hierarchical. A hierarchical floorplan is one which allows modules to consist either of cells or of other modules. A hierarchical floorplan can be represented by a tree, where nodes represent modules and the children of a node are the modules of which the parent is composed. A leaf node (or leaf cell) is a module which consists of a single component and might be a single gate or composite gate from a standard cell library or even a PLA. Strictly speaking, the structure used to describe hierarchical flooplans is not a general tree, but is a directed acyclical graph (DAG). However, a DAG doesn't give the impression of hierarchy which is so important to our floorplanner. On the other hand, nodes in trees are not supposed to have more than one parent. In the hierarchy used by FLINT, modules which are components of other modules have several parents, so that if module ***A*** is a component of modules ***B***, ***C***, and ***D***, then ***A*** has ***B***, ***C***, and ***D*** as parents. Since the term tree is more descriptive of the internal structure used to describe the circuit hierarchy, this is the term that shall be used throughout this document to describe the hierarchical internal structure.

The term floorplan usually refers to the floorplan of the entire circuit, but for lack of a better definition, it will also be used throughout this chapter to refer to the general physical arrangement of the components within any module. If a distinction needs to be made, it will be clear from the context which one is being referred to. The term block is also used throughout this chapter. A block is any group of components separated from the rest of the circuit or layout boundaries by four slices. Thus if we say that a module is composed of four blocks, we mean that the module has four distinct groupings of components separated by three slices. If the floorplan in Figure 3.1 were a module, then it would be composed of four blocks, **A**, **B**, **C**, and **D** separated by the three slices. Each of these blocks is composed of two more blocks, which happen to be components in this example, but could be made of further blocks. The term "module definition" will refer to the partlist of a module, the interconnections of those parts, and the names of external connections obtained from the schematics. Two modules with identical definitions must have the same partlists, interconnections, and external connections, making them interchangeable, provided the external connections are properly made.

3.2 Design Objectives

Identifying objectives is essential in creating an effective floorplanner. The most important decision, and one which bears directly on identifying other objectives, is whether FLINT should be totally automatic or interactive. We chose to make an interactive floorplanner since, to date, human creativity still seems to produce the best floorplans. Many objectives governing the design of FLINT followed from this decision. An interactive floorplanning tool should be flexible enough to allow the design of any part of the floorplan at any time, but should avoid displaying unnecessary detail. Treating the floorplan as a multi-level hierarchy goes a long way towards accomplishing this objective, since this closely coincides with actual circuit descriptions, and also provides for a built-in divide and conquer approach to floorplanning.

The designer should have a significant amount of freedom in floorplanning the circuit including everything from placing each component of a module, to completely rearranging the floorplan hierarchy. At the same time, FLINT should treat modules with identical definitions similarly so the designer doesn't have to continuously re-invent the wheel. For example, when laying out several identical instances of a module next to each other (such as the individual bits of a RAM, or the bits of a register), FLINT should group them together and treat them identically.

FLINT should allow the designer to propose several alternate floorplans of a module in his quest to discover one which fits in well with the total floorplan. The decision as to which alternative module floorplan to incorporate into the design should be delayed as long as possible so that the most informed choice can be made. Allowing the designer to propose new alternate module floorplans when the floorplan is near completion allows him to take more global information into account. A given alternate floorplan for a module should be available to all instances of the module and not be restricted to the instance for which it was created. FLINT should be able to select the alternative which will result in the smallest floorplan for each instance. The designer should still have control over such floorplanning parameters as, the aspect ratio of floorplan, being able to set bounds that FLINT must meet in its selection of floorplan alternatives. Furthermore, FLINT should allow fixed blocks to be incorporated in the floorplan.

Immediate feedback should be available as to the effects changes have on the floorplan. Perhaps the most effective way of accomplishing this is to include a pictorial representation of that part of the circuit currently being laid out and to have that representation change as changes are made to the floorplan. Including statistics, which compare the floorplan before and after changes, would also aid the designer in evaluating the effects of the changes.

Finally, an interactive tool should allow the addition of future algorithms to drive the floorplanning process. This would still allow an automatically generated floorplan to be fine-tuned manually.

3.3 Overview of Features and Limitations

Many of the objectives were successfully met by the new floorplanning utility. The flooplanner represents hierarchical floorplans with slicing structures. It creates the initial multi-level hierarchy from the schematic, using the same module names as used by the schematic, so the designer has a consistent reference to modules. FLINT uses a graphical user interface to aid in interactively floorplanning circuits. The designer can floorplan modules in any desired order, moving freely about the floorplan hierarchy. The display only shows the components of the module currently being floorplanned. Displaying all the cells of a large circuit would make floorplanning very difficult. Because of the hierarchical circuit definition, the components of the module displayed are the module instances at the next lowest level in the hierarchy. This hides the underlying module definitions until they are themselves floorplanned. There are commands available to modify the floorplan which are accessed via menus using a mouse driven input system and a point-and-click method of selection. Any of the floorplanning commands can be used to floorplan the module on display.

FLINT uses common definitions for identical module definitions, which appear throughout the design, to minimize the amount of floorplanning which needs to be done. It uses iterated structures to represent identical modules which are spatially adjacent and were identified as iterated structures from the schematic diagrams. Floorplans may be composed of a mixture of fixed and flexible blocks. FLINT allows the design of multiple alternative module floorplans and, at the end of the floorplanning process, optimally chooses the one which best meets the design criteria of size and aspect ratio. FLINT reports on the effects which changes made to the floorplan have on the overall design, so the designer can tell whether or not his ideas are having the desired effect. If necessary, FLINT is flexible enough to allow the designer to significantly modify the hierarchy, but since the hierarchy is obtained from the schematic diagrams, the need for extensive rearrangement is unlikely.

To help the designer work effectively, feedback is immediately provided on the effects of changes made to the floorplan. After the change is made to the floorplan, the area and shape of components are visibly changed to reflect the shape it should take on to make the floorplan on display as compact as possible. Along with the visual changes, statistics about the floorplan are also updated after each change. Statistics are available on floorplan area, areas and x and y dimensions of the module and whole circuit floorplans, and whether the change increased or decreased the total floorplan area.

There are a variety of operations available for modifying the floorplan. The designer may create a new alternate floorplan for a module at any time using the floorplan on display as a template. He may then look at any alternative and change its floorplan independently of the other alternatives. Changes to a floorplan alternative affect that alternative for all instances of that module, so the designer doesn't have to perform identical placement operations for each instance of that module in the hierarchy. Extensive rearrangement to the floorplan can be done by creating new levels or deleting existing levels in the hierarchy and by moving module instances from one level to another. The operations for modifying the floorplan of a single module instance are discussed later in this chapter.

Many circuits contain functional blocks which are made up of identical modules that are laid out next to each other. For example, all the bits of an adder or a register are identical and are located next to each other. We have created a new module type called an iterated structure to simplify dealing with blocks of this type. An iterated structure is similar to a separate module instance, but there are important differences between the two. The modules of an iterated structure must be laid out next to each other, either vertically or horizontally, whereas module instances may be located anywhere in the floorplan. The iterations (modules) in an iterated structure have identical floorplans, whereas each module instance may use a different alternative floorplan. The modules of an iterated structure may have alternate floorplans available, but the floorplan chosen for one element in the iterated structure is the floorplan chosen for all. Since this may prove to be too restrictive, iterated structures may be broken apart and groups of elements treated separately. Iterated structures are identified in the schematic and are adopted for use by FLINT.

Since no module generation utilities are presently available at the University of

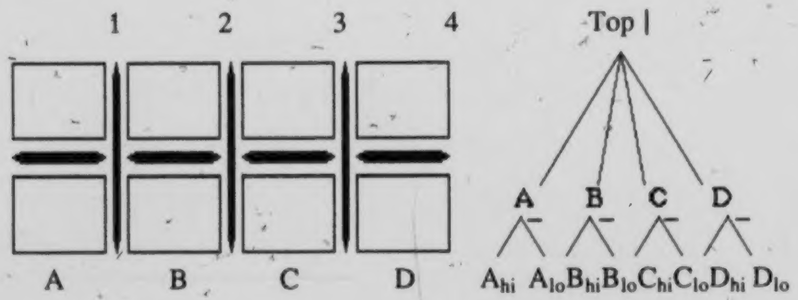
Alberta, floorplanning with flexible blocks is not possible. To avoid becoming a strict macro-cell placement program, we reached a compromise. Modules at the lowest level in the hierarchy, those composed only of cells, may have several alternative floorplans from which FLINT can choose. Instead of having a continuous curve of possible shapes and areas for each module, it allows several discrete shapes. The program for the floorplanner is modular, so when module generators and functions describing module area become available, incorporating these into FLINT should be easy. The modular program also makes upgrades and modifications or the addition of algorithms to drive the floorplanning process easier.

3.4 Introduction to Slicing Structures

One more design point should be made before discussing the operation of FLINT. FLINT uses slicing structures rather than general floorplans. Several factors contributed to this decision. First a multi-level floorplan hierarchy was chosen, and since the hierarchy itself can be represented using a tree, as can slicing structures, this choice allows common handling of the hierarchy and the floorplanning of modules. Second Stockmeyer's algorithm [Stock83] for optimal module orientation only applies to slicing structures, and we wished to adapt this for use with multiple alternate module floorplans. Furthermore, as Stockmeyer pointed out, the orientation algorithm for general floorplans is **NP**-complete, and so there is no known efficient algorithm for the general case. Finally, operations for manipulating a slicing floorplan appear to more straightforward than those for manipulating a general floorplan.

The discussions on the multi-level hierarchy in this thesis use the same terms used when discussing trees in computing science. A module consisting of components can be referred to in familial terms, where a module is a parent, the components are its children. Generally speaking, a component could be a module instance which, in turn, consists of components or, in these new terms, a parent with its own children. Using this terminology the statements "module's child" or "module's parent" should be clear: The first case refers to a component of a module; the second to another module which has the given module as one of its components.

Like modules and components, hierarchical treatment of slices is possible. Each slice separates a block of modules in two. In Figure 3.1a the vertical slices separate the block into four sub-blocks, while the horizontal slices separate each of the four sub-blocks into two sub-sub-blocks. The horizontal slices are lower in the hierarchy than the vertical slices, and can be thought of as the children of the vertical slices as shown in Figure 3.1b.


1
2
3
4
A
B
C
D
Top |
A
B
C
D
A_{hi} A_{lo} B_{hi} B_{lo} C_{hi} C_{lo} D_{hi} D_{lo}

a

b

Figure 3.1: Slicing Structure and Slicing Tree

If the numbers 1-3 each represent a slice and the letters A-D each represent a block, then slices 1-3 separate blocks A-D (slice 4 separates the module from an adjacent module, or from the IC boundary and is not shown). The slice | at node ***Top*** in the tree represents all four vertical slices. In Figure 3.1, the slices 1,2, and 3 are equal-level siblings. The blocks A, B, C, and D, are also siblings. The pair of blocks A_{lo} and

A_{hi} which make up block A are also siblings. Associations between slice directions and nodes in the hierarchy tree are stored with each node and are shown as horizontal or vertical bars in the figure. When blocks are separated vertically, the left to right order of nodes in the tree determines the left to right order of blocks in a floorplan. When separated horizontally, the left most block in the tree is the highest block in the floorplan.

3.5 Floorplanner Operations

FLINT offers many features which aid the designer in creating new design alternatives, changing the graphical view of the circuit, and manipulating floorplans and structures.

3.5.1 Alternative Creation and Deletion

The first operations presented concern creating and deleting alternatives. These operations are very straightforward. Creating an alternative for the module on display only requires invoking the ***create_alternative*** operation. The newly created alternative has the same floorplan as the one displayed. FLINT places no restrictions on the number of alternatives which can be created. ***Delete_alternative*** is also straightforward. Selecting the operation deletes the alternative on display. FLINT won't allow the deletion of the last remaining alternative.

3.5.2 Circuit Viewing

The designer can use several different floorplanner operations to affect the floorplan: however, in order to evaluate the results and choose the next operation appropriately he must be able to view various parts of the floorplan. This is done with the ***change_view*** operation. With this operation, the designer can make any module appear on the display.

Change_view selects modules for display in an orderly fashion, following the circuit hierarchy. Only the components of one module are displayed at a time. If the designer is viewing module A, shown in Figure 3.2a, he has three choices of what to display next. He can display one of module A's components, as a module, he can display A's parent, showing A as a component, or he can display one of A's alternatate floorplans. If the next module displayed is A's parent, then we say that the display has moved up one level in the hierarchy. If the next module displayed is one of A's components, then we say that the display has moved down a level in the hierarchy. If the next module displayed is one of A's alternatives, then we have not changed levels in the hierarchy.

To display one of A's components as a module, module B for example, we select the ***change_view*** menu item, select module B, and click the left-most mouse button. The display will change to show the current floorplan for module B, as in Figure 3.2b.

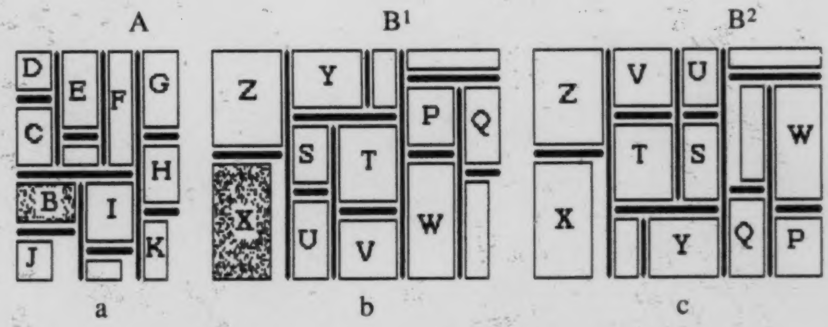


Figure 3.2: Change Circuit View

To redisplay module A, or the parent of any module displayed, we select the ***change_view*** menu option and click the middle mouse button. The ***change_view*** function doesn't end until a sub-menu option is chosen. Once the ***change_view*** option is invoked, the designer can move up or down as many levels in the hierarchy as desired by clicking the left and middle mouse buttons. Alternate floorplans of a module may be viewed through a second sub-menu option. To view the alternate floorplan of B, shown in Figure 3.2c, select the ***change_view*** function, then the view alternative sub-menu function, and click the left or middle mouse buttons. Alternative floorplans are stored in a circular linked list. The leftmouse button cycles forward through the list, while the middle mouse button cycles backward through the list.

3.5.3 Operations Affecting Module Floorplans

Several operations are available for changing the floorplan of a module by rearranging its components. These include: 1) changing the slice direction separating modules, 2) changing the order of blocks which are the children of a particular node, 3) reversing the order of blocks which are the children of a slice, and 4) moving blocks to different locations within the module. Using these operations, the components of the module can be laid out in any possible slicing configuration. We shall discuss these operations in this order.

If all the slice directions were the same, the components of a module would be laid out in a straight line. ***Change_slice_direction*** splits a row of blocks into two groups and separates them with a slice of the opposite direction. This operation rearranges the slice tree in a way that can't be undone with a single operation. Correction requires a series of ***change_slice_direction*** operations. Changing the slice direction of slice 2 in

Figure 3.1 would result in the floorplan shown in Figure 3.3, while changing the vertical slice between the blocks of B in Figure 3.1 would result in the floorplan of Figure 3.4.

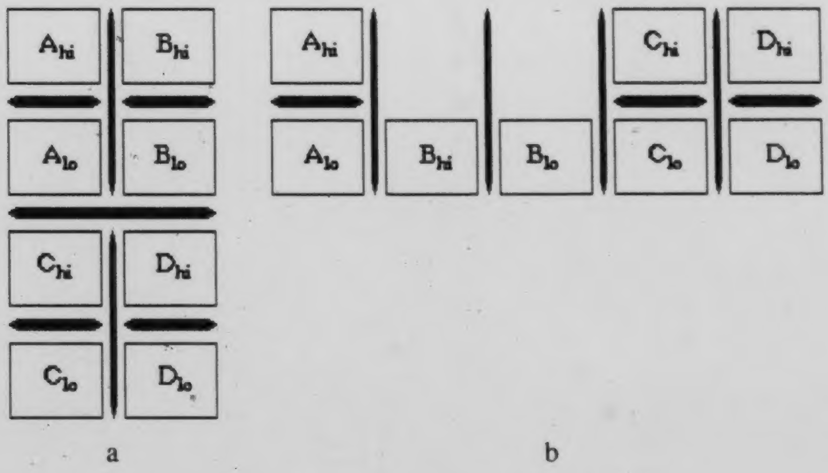


Figure 3.3: Changing Slice Directions

The order of sibling blocks may be changed by moving them, one at a time, with the ***change_order*** function. This operation takes two selections, moving the first block selected in front of the second block selected. FLINT only allows siblings of the first block chosen to selected as the second block, since this operation only works on siblings. In the example of Figure 3.4a the first block selected is ***D*** followed by block ***B***. The result is the floorplan shown in Figure 3.4b.

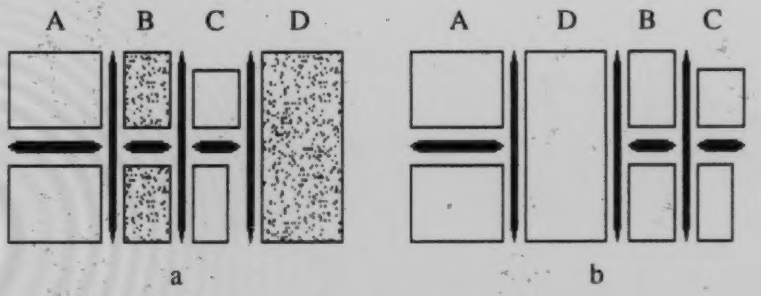


Figure 3.4: Changing the Order of Children

Change_order may also be used to move a block to the end of a row. This is done by selecting the same block twice. Selecting block ***B***, as in Figure 3.5a, results in the floorplan shown in Figure 3.5b.

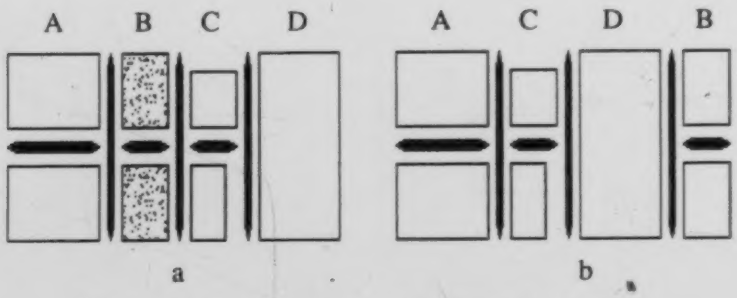


Figure 3.5: Moving a Block to the End of a Row

The order of the children of a block can be reversed, if this is more prudent than changing their order one at a time. ***Reverse_order*** involves all the children of the block. In Figure 3.6a the lightly dithered block has the order of its children reversed resulting in the floorplan of Figure 3.6b.

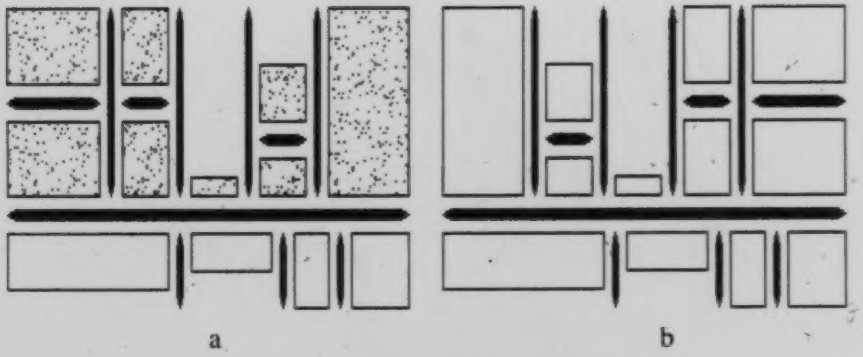


Figure 3.6: Reverse Order of Children

The most powerful operation for changing floorplans is ***cut_and_paste***. Any block can be cut out of its present location and reinserted in another location in the floorplan. The operation of ***cut_and_paste*** is similar to that of block movement, except that the second block selected can be anywhere in the floorplan. ***Cut_and_paste*** between modules is discussed later; for now, assume that both selected blocks are in the same module. One limitation of ***cut_and_paste*** is that blocks can't be inserted at the end of a row. They must be inserted elsewhere in the row, and then moved to the end of the row using block movement. A ***cut_and_paste*** example is shown in Figure 3.7. In Figure 3.7a the first selection is the lightly dithered block, the second selection is the heavily dithered block. The first block is inserted ahead of the second resulting in the new floorplan shown in Figure 3.7b. ***Cut_and_paste*** does nothing to a single module which can't be done with the other operations. Its real power appears when used across levels of the hierarchy.

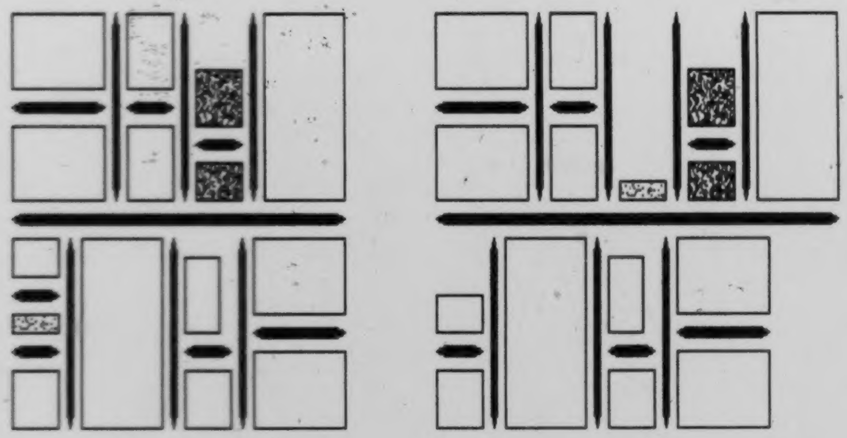


Figure 3.7: Cut and Paste Within the Same Module

The ***change_order*** operation, discussed earlier, is a subset of ***cut_and_paste*** operation; thus the second block selected is constrained to be a sibling of the first block selected. This makes selection easier, an advantage over the ***cut_and_paste*** operation, although it is limited in its range of operation. Being able to move blocks to the end of a row is another advantage of this operation. ***Cut_and_paste*** can't paste blocks to the end of rows, they must be inserted elsewhere and moved to the end of the row with this operation.

3.5.4 Operations on Iterated Structures

All the operations discussed so far are available for use with all module blocks. Iterated structures have two operations all to themselves. Before introducing these operations, something more should be said on iterated structures. Iterated structures are, in fact, one dimensional, but they may be stacked to appear multi-dimensional. To do this, each member of the iterated structure at the highest level in the hierarchy tree could have as its only component another iterated structure. If the highest level of the iterated structure consists of four iterations, and the next level consists of eight iterations, then the whole structure appears to be a 4*8*1 structure; thus a change to the floorplan of one component changes all 32 components.

If the iterated structure created from the schematic is inconvenient, two operations which affect only iterated structures are available to alter it. The first operation splits a structure into two parts. The second interleaves the iterations of two identically sized structures.

Split_structure splits an iterated structure into two parts allowing each to be treated independently. There is no restriction on the relative size each part must take after being split. Thus a structure of size 8 can be split into parts with sizes 1 and 7, 2 and 6, 3 and 5 etc.. The designer may continue splitting the parts until each contains only one module. Once split, either part may be relocated anywhere within the circuit floorplan, and the floorplans of each part may be modified independently. In the example in Figure 3.8, the highlighted slice of Figure 3.8a shows where the split will occur; between the 4th and 5th iterations of the module.

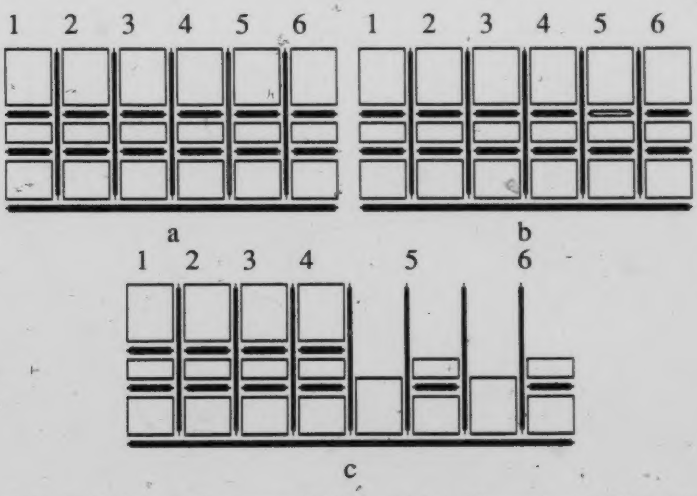


Figure 3.8: Splitting a Structure Followed by Changing a Slice Direction

The floorplan doesn't change after the operation. After using ***split_structure***, changing the slice direction of the highlighted slice shown in Figure 3.8b, results in the floorplan of Figure 3.8c. Note that ***change_slice_direction*** affects both iterations 5 and 6 of Figure 3.8b, since they hadn't been split from each other, but doesn't affect iterations 1 through 4.

The other operation affecting iterated structures is ***interleave***. This operation combines a pair of identically sized iterated structures into one, by interleaving their iterations. This operation requires the selection of two adjacent iterated structures. Interleaving the members of the structures occurs at the highest level, and so they must have the same number of iterations at that level. Interleaving a 4*8*1 structure with a 4*1 structure is possible, but not a 4*8*1 structure with a 2*16*1 structure. In the example shown in Figure 3.9a two iterated structures, *a* and *b*, are selected for interleaving. The resulting floorplan, shown in Figure 3.9b, is a single iterated structure alternating blocks of *a* and *b*.

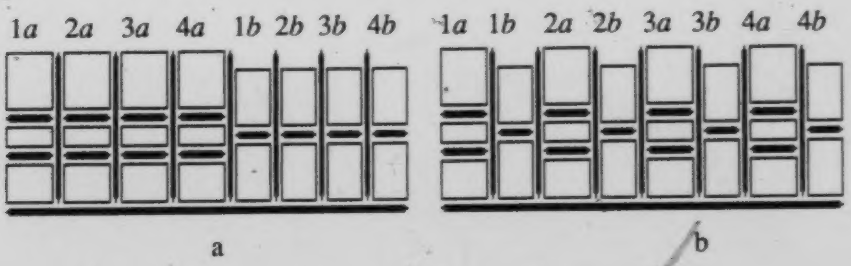

1*a* 2*a* 3*a* 4*a* 1*b* 2*b* 3*b* 4*b*
1*a* 1*b* 2*a* 2*b* 3*a* 3*b* 4*a* 4*b*
a
b

Figure 3.9: Interleaving Iterated Structures

3.5.5 Operations Affecting Circuit Hierarchy

Operations affecting the circuit floorplan aren't restricted to rearranging the components of modules. Modification of the circuit hierarchy to increase floorplan flexibility is possible. There are three ways to rearrange the circuit hierarchy: 1) flatten the circuit hierarchy, 2) combine several components to create a new module, 3) move modules to different locations in the hierarchy. This third operation is discussed separately in section 3.5.5.1.

The first of these, ***flatten***, removes a module record from the circuit hierarchy, but not the instance records which make it up. The components (identified by the instance records) which made up the module definition become components of its parent. The example in Figure 3.10 shows how ***flatten*** works. Figure 3.10a shows module ***A***, the parent of module ***B*** (which is highlighted) who's definition will be removed by ***flatten***. Figure 3.10c shows how ***A*** would look after the definition removal.

Note that the components which previously defined module ***B*** in Figure 3.10b now take ***B***'s place in module ***A***. Because ***flatten*** is not reversible on modules in the original hierarchy, a module alternative is created, identical to this one, and the changes are implemented on the alternative. If the designer really wishes these changes to be the only choice, he can remove the original alternative.

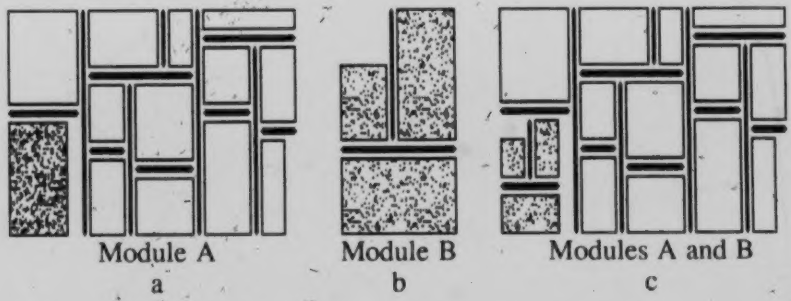


Figure 3.10: Flattening the Circuit Hierarchy

The removal of the module definition only affects this instance module ***B***. Other alternatives of module ***A*** remain unaffected, and still have module ***B*** as one of their components. All instances of module ***A*** in the floorplan have the floorplan alternative shown in Figure 3.10a replaced by the floorplan shown in Figure 3.10c.

Creating a new level in the hierarchy requires selecting one block of components which is to become the new module. FLINT provides a default name for the new module if the designer doesn't explicitly provide one. Creating a new level from the highlighted block of Figure 3.10c changes module ***A*** into the module of Figure 3.10a. The new module, named ***B***, is shown in its entirety in Figure 3.10b and is shown in Figure 3.10a as the dithered block. The new module can then be treated just like any other module in the hierarchy.

It is important to realize that removing a module definition from the hierarchy permanently changes the hierarchy and often can't be reversed by creating a new level with the same components and the same name. If several modules contain instances of module ***B***, in the above example, and the definition of module ***B*** is removed from the hierarchy, a copy of ***B***'s components is made and they become components of ***A***. Grouping these components together into a new level and naming it ***B*** will result in module ***A*** containing a unique instance of a module ***B***, say $\boldsymbol{B}^*$. No changes to the floorplan of $\boldsymbol{B}^*$ will affect modules which contain original instances of ***B***.

3.5.5.1 Operations Affecting Circuit Hierarchy and Placement

As stated earlier, the ***cut_and_paste*** operation can be used to move modules to other levels in the hierarchy. Cutting and pasting between modules is done in much the same way as cutting and pasting within a module. The cut block is selected first, but before the paste block is selected, the ***change_view*** operation is selected to change modules. Once the desired module is entered, ***change_view*** is exited, and a paste block selected. The cut block is then pasted next to the paste block. Cutting and pasting can be done from any module to any other module in the hierarchy, but there are some restrictions. First, ***cut_and_paste*** won't work between module alternatives, since this would alter the circuit definition. Second, the hierarchy can't be restored once a module's location in the hierarchy is changed, even though the module can be moved back to its original location. To see why this must be so, consider a move such as the one shown in Figure 3.11, without worrying about how the move is made. In this example, module ***a***, shown in Figure 3.11a, is to be cut from its location in module ***X*** and pasted into a new location in module ***b***, shown in Figure 3.11b. After the move, module ***X*** will appear as in Figure 3.11c and module ***a*** will appear as part of module ***b*** in Figure 3.11d. To prevent any other modules in the hierarchy with instances of module ***b*** as one of their components from suddenly acquiring module ***a*** as part of their definition, a unique copy of module ***b***, ***b****' is created for module ***X***, when ***a*** is moved into it. If the operation is now reversed by moving ***a*** back to its original location in module ***X***, module ***b****' will still be unique to module ***X*** and no changes to its floorplan will affect modules other than ***X***.

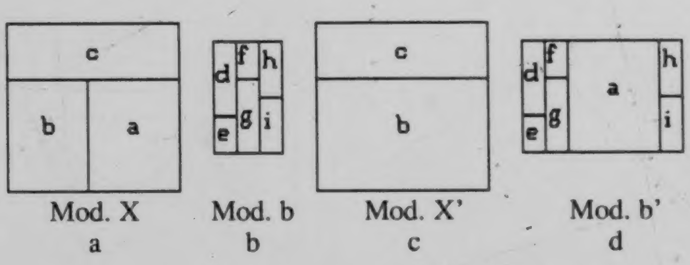


Figure 3.11: Cut and Paste Across Hierarchy Levels

Third, the designer may not move modules into instances of themselves. For example, if modules ***a*** and ***b*** in the example of Figure 3.11 had been instances of the same module, FLINT would not have allowed the operation. The designer can work around this by first removing ***a***'s definition, so that its components become part of module ***X***'s level and then cutting out all the components which used to belong to ***a*** and pasting them into the floorplan of module ***b***.

Other restrictions to ***cut_and_paste*** concern iterated structures. First, cutting and pasting between iterated structures requires their array sizes to be identical. Second, moving blocks from one multi-dimensional structure to another requires the array sizes of the structures to be identical in every dimension. Finally, moving an instance between iterations of the same structure can only be done after the iterated structure has been split (as discussed earlier) into parts of equal size.

The restrictions on ***cut_and_paste*** are in place to prevent adding or removing cells from the circuit definition. Because cutting and pasting modules across levels can have such wide ranging effects, FLINT creates an alternative using the original hierarchy, as a recovery measure. The designer can undo the changes by deleting the alternative created by the ***cut_and_paste*** operation.

3.6 Other Features Under Designer Control

Another feature under user control which should be discussed is aspect ratio. This is not an operation, but still affects the final floorplan. An aspect ratio limitation may be specified when the program is run. FLINT will choose the floorplan with the smallest area which doesn't exceed this limitation, if possible. If it isn't possible, FLINT chooses the floorplan which comes closest to meeting the aspect ratio criteria. The aspect ratio specified by the user is a number between 0.0 and 1.0. This number represents the length of the shorter side of the layout divided by the length of the longer side. An aspect ratio of 1.0 defines a square layout . FLINT will always try to create the floorplan whose aspect ratio is greater than or equal to the aspect ratio provided by the user and whose area is minimal. If the aspect ratio criteria can't be met, then FLINT will create the floorplan whose aspect ratio is closest to 1.0, regardless of size.

3.7 Features Under Floorplanner Control

After all the floorplan work is done, there are still several module floorplan choices to be made. These are the choices of which floorplan alternatives of the modules to use in creating the final floorplan. This choice is entirely up to FLINT.

Should the designer decide not to use some alternatives in some instances, his only recourse is to create unique module instances and delete the alternatives he doesn't want used. He can do this by deleting module definitions, and then creating new levels in the hierarchy. As previously mentioned, this creates unique module instances. This could be a very time consuming if it had to be done for a large number of modules.

This concludes the discussion of the user controlled operation of FLINT. With this set of operations, any slicing floorplan can be created.

Chapter Four: Implementation Details

This chapter presents information pertinent to the internal operation of FLINT, beginning with a description of the main data structures used in the floorplan, how these structures connect, and the fields they contain. The second section discusses the algorithms used to implement many of the features presented in Chapter 3. Straightforward algorithms, graphics algorithms, and input algorithms are not included in this discussion. The third section of this chapter discusses some of the software tools which form an integral part of the floorplanning package.

4.1 Data Structures

4.1.1 Hierarchy Tree

Within the floorplanner, a multi-way tree stores complete layout information. There are four different types of nodes in a multi-way tree including module alternative nodes, instance nodes, iteration nodes and slicing nodes. The following paragraphs discuss the distinguishing traits of the nodes and how they are used in the internal data structures.

4.1.1.1 Module and Instance Records

This discussion of the hierarchy tree only considers three relevant fields in the record . The first of these fields is the name field, the second is the iteration count field and finally, the schematic name field. The purposes of these fields will be made clear in the ensuing discussion. We shall illustrate the use of the record types with a small example module. An example module ***A***, composed of four components, ***B***, ***C***, ***D***, and ***E*** is shown in Figure 4.1a where the four components are separated by vertical slices. The records which represent the floorplan for this module are shown if Figure 4.1b. The record named "A" is a module record. Module records always signify a new layer in the circuit hierarchy. They also have some special properties which will be discussed in greater detail later in this section. The four children of *A*, marked "B," "C," "D," and "E" are instance records. Each instance record either represents an instance of a module or an instance of a leaf cell.

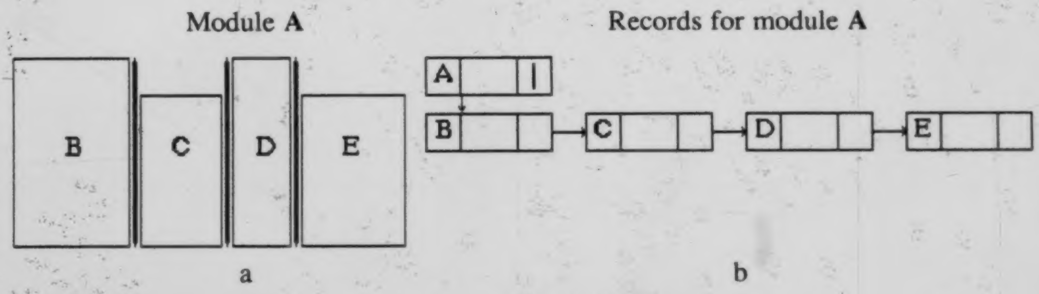


Figure 4.1: Simple Floorplan and Tree

An instance record does not represent the module (or leaf cell) itself, rather it points to a module or leaf cell record. The instance record itself suggests an occurrence of the leaf cell within the given module. Using the example of Figure 4.1, the record named "B" is an instance record, indicating that one of the components of module *A* is another module ***B***. If ***B*** were composed of three more components, x, y, and z, then the tree of records would look like that shown in Figure 4.2b, and the floorplan would look like that shown in Figure 4.2a. The components x, y, and z are shown in a dithered box to indicate that they belong to module ***B***, and are only indirectly part of module ***A***.

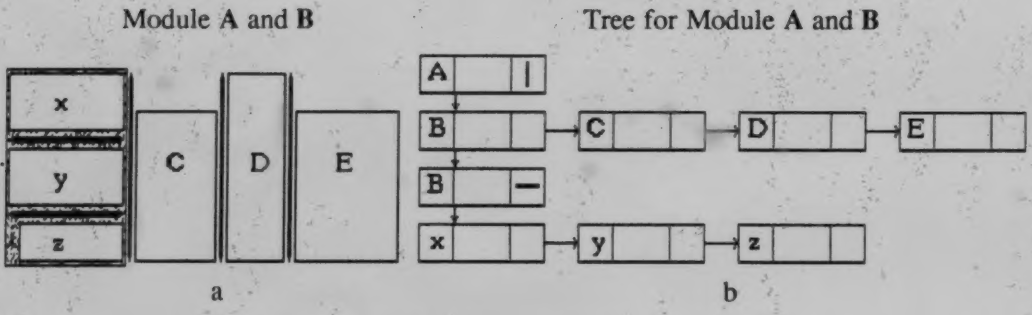


Figure 4.2: Two Level Floorplan and Tree

The right-most field in the module records ***A*** and ***B*** (as shown above) contain slice direction information. The vertical bar in record ***A*** indicates that a vertical slice separates its components, ***B***, ***C***, ***D***, and ***E***. The horizontal bar in module record ***B*** indicates that its components are separated by a horizontal slice. Instance records contain no slice direction because instance records that point to leaf cells do not need a slice direction and instance records that point to module records would just duplicate the slice information already contained in the module record.

4.1.1.2 Slice Direction Records

If module records were the only records which indicated slice direction, then FLINT would either produce very limited floorplans, or would have a tremendous number of levels in the hierarchy. To rectify this we employ a third type of record, the slicing record, in creating floorplans. Records of this type do not create new modules in the circuit hierarchy, but introduce new slices into a module's floorplan. To see how slicing records are used, assume we wish to separate module *A* of Figure 4.1a into two parts, separated by a horizontal slice, as in Figure 4.3a. The tree for this new floorplan is shown in Figure 4.3b. Note the addition of two new records and the change in module *A*'s slice direction. The new slice records indicate the slice direction separating their children is vertical. The horizontal slice in *A* indicates the blocks represented by the new slice records are separated horizontally. It is also worth noting that the linked list of Figure 4.1b which contained records for ***B***, ***C***, ***D***, and ***E*** has been split into two lists, one containing ***B*** and ***C***, the other containing ***D*** and ***E***.

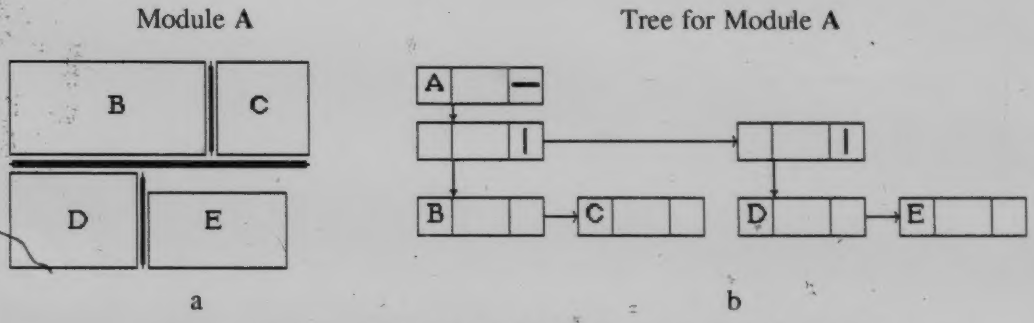


Figure 4.3: Floorplan of Two Blocks/Four Modules and Tree

If the floorplan record structure was truly a tree there would be no need to have instance nodes, since each would point to a single child. As stated early in Chapter 3, a floorplan is not stored as a true tree, rather it is a DAG, since the module nodes may have several parents. This is where instance nodes make their presence felt. Module nodes always have instance nodes as parents. A module named ***E*** has parents named ***E***. Furthermore a secondary linked list connects all instances which point to module ***E***, so that other modules which contain an instance of ***E*** can be easily discovered. From here on we shall refer to sibling elements of the linked lists first mentioned as sisters, and to those in this new linked list as brothers. Only instance records can have brothers, but all records can have sisters. In the example of Figure 4.4, modules ***A*** and ***H*** both contain an instance of module ***E***. The instance records named "E" both point to the same module record, and the instance records are linked in a circular list.

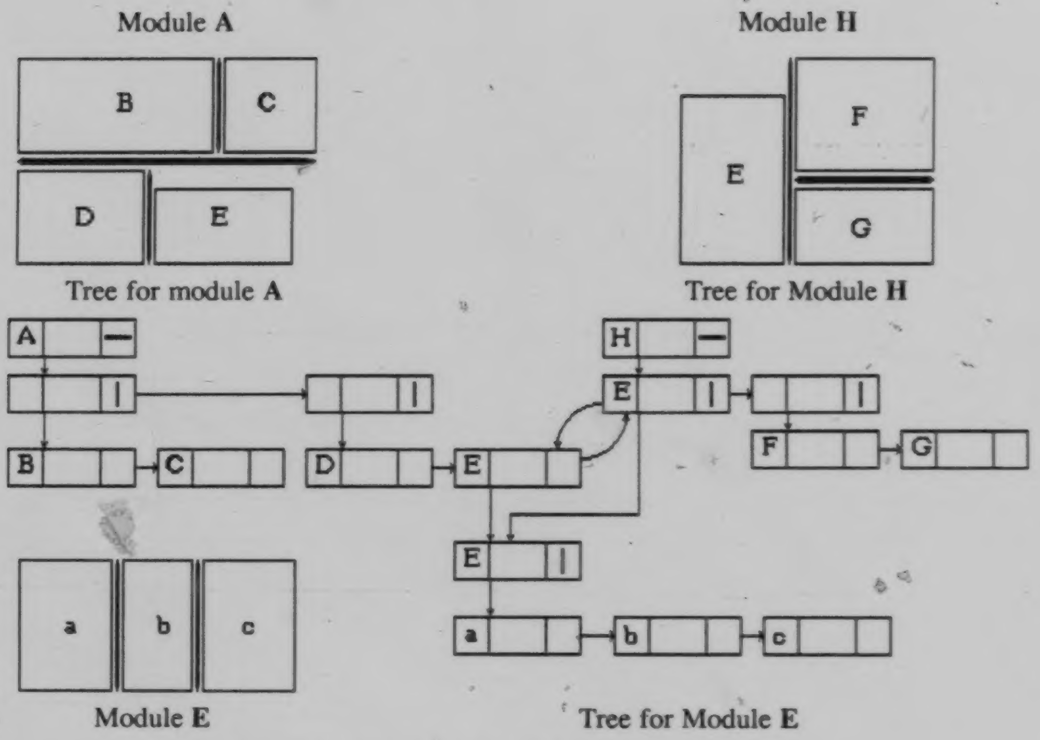


Figure 4.4: Two Modules With Common Instance

The relationship between module nodes and instance nodes does not end here. FLINT allows the designer to create several different floorplans for each module and, for each instance, it chooses from among the alternative floorplans the one which minimizes the amount of area consumed by the entire circuit floorplan. A linked list connects the alternative floorplans together. All instances point to the same module alternative, but FLINT may choose any alternative for any instance in the final design. Also the designer may select one of the alternatives and make changes to its floorplan. The example of Figure 4.5 shows module *E*, as in Figure 4.4, but with an alternative floorplan (much of the tree structure and other module floorplans shown in Figure 4.4, have been removed here). This example shows a tree with two instance records and two module records for module *E*. The alternative module records (named "E^1" and "E^2".

where the superscripts just distinguish the two) are connected in a linked list.

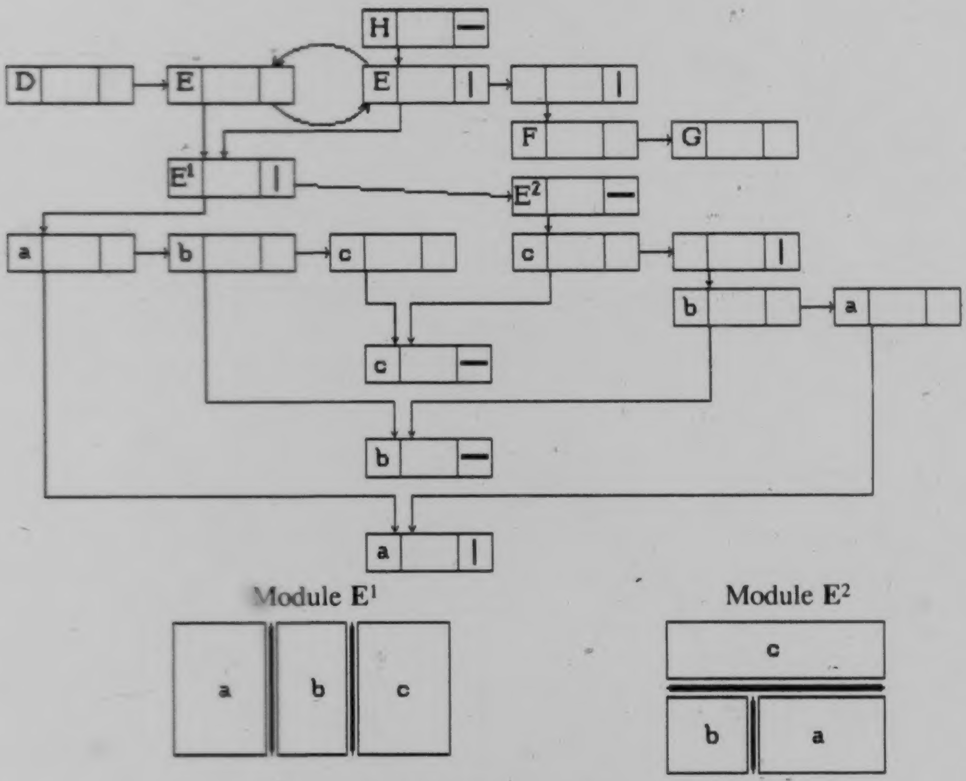


Figure 4.5: Tree With Two Alternatives

The instance records of both $\boldsymbol{E}^1$ and $\boldsymbol{E}^2$ point to the same leaf cell records, ***a***, ***b***, and ***c***. If ***a***, ***b***, and ***c*** were module records, they would also have children. The only sisters which module records may have are alternative module records. The instance records of alternative modules will converge to point to the same modules.

Although this pointer arrangement between instance and module nodes saves storage space when there are many instances of a module, this was not the primary reason for structuring floorplans this way. The main reason for using this type of structure is to allow the alternative floorplans created for a module to be available throughout the circuit floorplan, wherever instances of that module are found. This allows the designer to create several floorplans for a module the first time he encounters it without having to redo that work the next time he encounters the module. It also allows FLINT to do size calculations on a module once and use that information for all instances.

4.1.1.3 Iteration Records

One final record type not yet encountered is the iterated structure record. Records of this type are unique to iterated structures and can be identified by non-zero counts entered in the iteration count field. This field indicates the number of times the list of children appears. The iteration field consists of a start index number and an end index number, so the floorplanner knows which set of iterations it is dealing with, and whether the iterations are arranged in ascending or descending order. When the floorplan is first created from the schematics, iteration records always point to a single instance record, which in turn points to a module record. There are times when multi-dimensional iterated structures are desirable. In this circumstance the iteration record would have another iteration record as its child which would then point to an instance record. As an example of a single dimensioned iterated structure, Figure 4.6a shows module ***A***, as in Example 3a, but with instance ***B*** iterated three times. The tree of Figure 4.6b includes the iteration record which is the parent of instance ***B***. Assume that instance ***B*** points to module ***B*** which consists of instances ***x***, ***y***, and ***z***, as in Figure 4.2, and that we eliminate the instance record and module record for ***B*** (this is a legal operation called expanding a module).

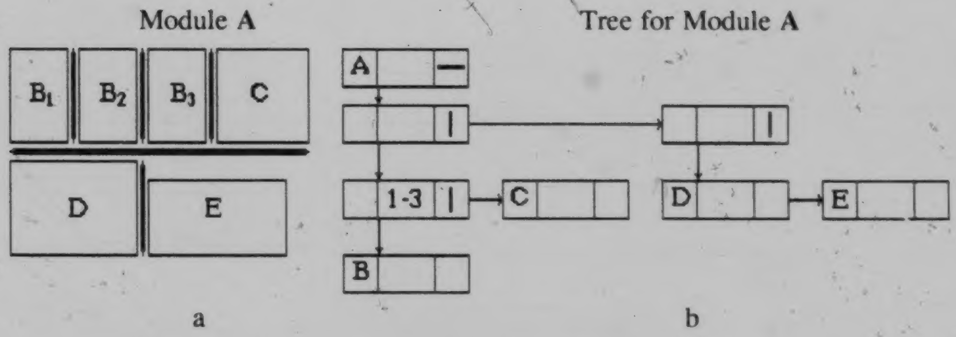

Module **A**
Tree for Module **A**
B1
B2
B3
C
D
E
A
—
|
1-3
|
C
B
|
D
E
a
b

Figure 4.6: Floorplan with Iterated Structure and Tree

The arrangement of the expanded module's components would appear as in Figure 4.7a, and the iteration record would point to the instance records ***x***, ***y***, and ***z***.

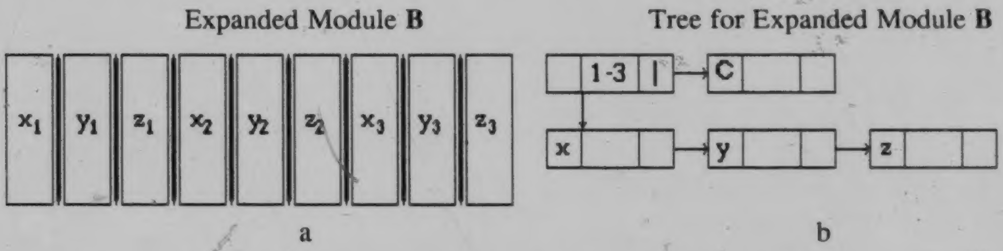

Expanded Module **B**
Tree for Expanded Module **B**
x1
y1
z1
x2
y2
z2
x3
y3
z3
1-3
|
C
x
y
z
a
b

Figure 4.7: Iterated Structure and Tree No Slice Node

Note in the floorplan of Figure 4.7a that vertical slices separate both instances (x, y, z), and the iterations (separating $<x_1, y_1, z_1>$,$<x_2, y_2, z_2>$, and $<x_3, y_3$, and $z_3>$). If we want to separate x, y, and z from each other with horizontal slices but still separate the iterations with vertical slices, as in Figure 4.8a, we need a slice record between the iteration record and the instance records of x, y, and z. This record (marked $\underline{S}$ in the tree of Figure 4.8a) only determines the slice direction separating the children of an iteration. The slice direction separating the iterations from each other is always stored in the module record. If the slice direction separating iterations is the same as the slice direction separating the children of the iteration, then the slice record (like **S**) is redundant and does not appear in the tree.

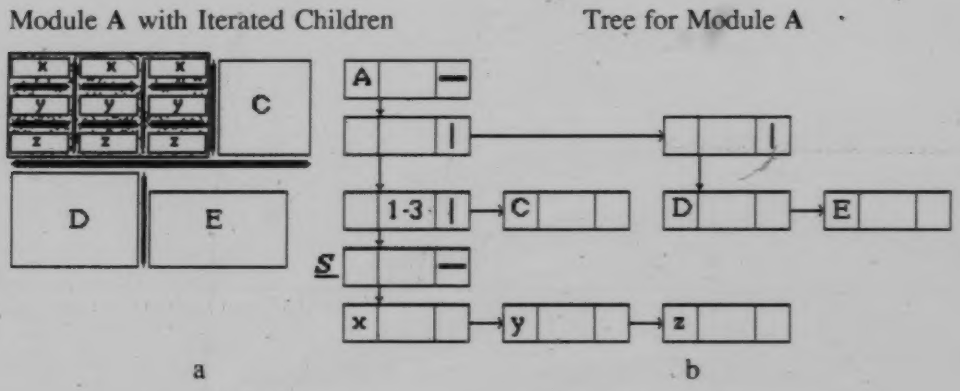


Figure 4.8: Iterated Structure and Tree With Slice Record

The last point is generally true of slice records. If a slice record and its parent have the same slice direction, then the slice record is redundant and FLINT removes it from the tree. This rule does not apply to module or iteration records. Since module records always have instance records as parents, and instance records contain no slice direction, module records are not removed from the tree. Iteration records contain counts not contained in module or slice records, and are not removed from the tree either.

4.1.1.4 Other Fields

Besides the pointers to the linked lists and children, records contain several other fields. All nodes in the tree use the same type of record and the field "terminal" distinguishes from among them. Each record represents a block whose coordinates are given in the fields "upper" and "lower". These coordinates are only used as a convenience in updating the display and are not crucial to floorplanning. The coordinates of the endpoints of a slice are given in the fields "slice_hi" and "slice_lo" and describe the slice separating this block from the block to the right (if the slice direction separating them is vertical) or to the block below (if the slice is horizontal). Records store slice direction in the field "cut_dir". The name of a module is given in the field "name" of module records. The name of the instance which points to a module is given in the field "sch_name" of instance records and contains the name given to the instance when it was created in the schematic. The name should describe the type of module (i.e. adder), while the sch_name should describe its purpose (i.e. add_offset_to_program_counter). Further, "sch_name" may include a hierarchical path, discussed later in this chapter. Iteration records use the fields "iter_start" and "iter_end" to store beginning and ending iteration indices of an iterated structure. This field must be more than a count, since structures can be split and FLINT must be able to distinguish between them. The iter_start is always the upper or left most iteration and iter_end is the lower or right most iteration.

4.2 Algorithms

This section presents some algorithmic design details of FLINT. The functions used by FLINT fall into four categories. Those which are external to the floorplanning program, those which perform graphics output or mouse driven input, support functions which are either trivial or straightforward and, finally, those which are important to the operation of FLINT and may require some explanation. Generally only functions which are members of the fourth category will be presented. With the material so far presented, some of the functions which operate directly on the tree become trivial to implement. As an example, reversing the order of children of a node, which reverses the order of appearance of the blocks in the floorplan and on the screen requires only exchanging the order of the children of a list after identifying it. Functions such as this may be mentioned without presenting details of the design.

4.2.1 **Support Routines**

The first group of functions discussed are important to the operation of FLINT because they are used throughout by other functions to operate on the tree. Several operations use the routine *Copy_Tree* to duplicate part of the tree. *Copy_Tree* duplicates the tree from the given record down as far as the instance records below it. Invoking this routine with a module record creates a copy of the entire module. The instance records at the bottom will end up pointing at the same module records which the instance records in the original module pointed to. The instance records are also added to the secondary linked lists of instance records, as previously shown in Figure 4.5. The modules which the instance records point to will point back to the new instance copies as their parents.

Another often used routine, *Clean_Up*, removes redundant nodes from the layout tree of a module. This involves removing slice nodes which have the same slice direction as their parents and are not iteration nodes. The example of Figure 4.2 shows a module tree before and after a clean up operation. In the example, *Clean_Up* removes node *a* from the linked list and replaces it with node C^i because node *a* had only one child, making *a* redundant. Nodes F^i and G^i replace node *b* because *b* has the same slice direction as its parent, making *b* redundant. The layout is the same for both these trees. Many of the operations leave redundant nodes in the tree, and use this routine to remove them.

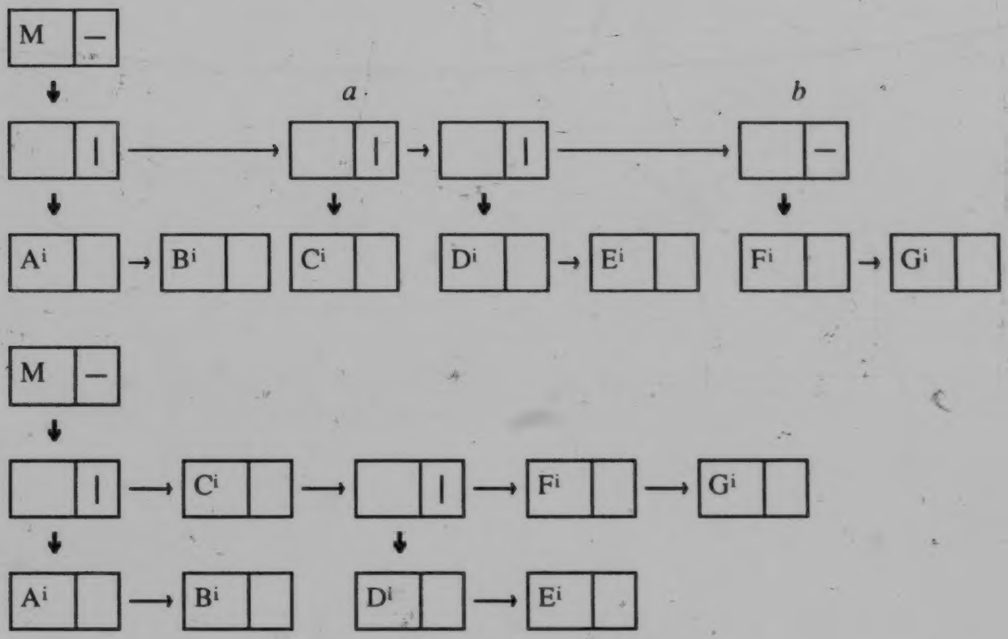


Figure 4.9: Clean Up of Tree

Several functions exist to insert and remove records from trees.

1. *Delete* removes a record from a linked list. The record is not disconnected from its offspring, so its offsping are removed from the module definition as well.
2. *Remove* also removes a record from a linked list, but the removed record is also disconnected from its sub-tree. Since this operation is only used on instance nodes, and their children have multiple parents, the sub-tree is not lost.
3. ***Insert*** **(*b*, *a*)** inserts record ***a*** into a linked list as a sister of record ***b***, just ahead of record ***b*** in the list. If record ***b*** were the first record in the linked list before the insertion, record ***a*** would become the first in the list after the insertion followed by record ***b***.

4. *Add* (***a***, ***b***) adds record ***a*** as a child of record ***b*** at the end of ***b***'s linked list of children.
5. *Split* (***a***) splits the list in which record ***a*** exists into two parts; from the beginning of the list to the node just ahead of ***a*** and from ***a*** to the end of the list. The first part of the list remains with the parent. Starting with ***a***, *Split* returns the rest of the list intact, but without a parent. If ***a*** is at the front of the list, *Split* removes the whole list.
6. *Spadd* (***b***, ***a***) adds the list headed by ***a*** to the end of the list of children of ***b***.

4.2.2 Operation Routines

The following algorithms perform operations which are executed by the user from menus.

From the designers point of view the procedure *Split_Node*, whose algorithm follows, just splits a structure into two parts. Internally *Split_Node* makes a copy of the tree from the iterated structure node down to its instance nodes, inserts the new copy as a sister of the original copy, and updates the iteration counts of the old and new iteration records. In the algorithm, the procedure *Check_children* checks to see if any of the instance nodes belonging to the iterated structure originated elsewhere in the tree, as might happen when a Cut and Paste operation moves children from one set of iterations to another set. If this had happened then *Split_Node* will abort.

```
Split_Node (iter_node, old_start, old_end, new_start, new_end )

/*    iter_node -- is the iterated node to be split in two
```

```
          old_start -- the starting iteration number prior to the split.
          old_end -- the ending iteration number prior to the split.
          new_start -- the starting iteration of the new iterated structure.
          new_end -- the ending iteration of the old iterated structure.
          (Note that new end will be either 1 greater or 1 less than
                           new_start) */

      BEGIN
      Check_children and return if operation is illegal;
      Allocate (new_iter_node identical to iter_node);
      Copy_Tree(new_iter_node, iter_node);
      Allocate (temp_node);
      Insert (temp_node as sister to iter_node);
      Delete (iter_node);
      Add (iter_node as child of temp_node);
      Insert (new_iter_node as sister of iter_node);
      iter_node  end_index = new_end;
      new_iter_node  start_index = new_start;
      Change path indices of sch_name for each instance in both new
                           structures;
      Clean_Up unneeded records.
      END;
```

Splitting an iterated structure creates two iterated structures, side by side, identical except in the iteration indices. The sum of iterations of the two new structures equals the number of iterations of the original structure from which they were created.

Flattening the hierarchy requires removing an instance and module node, and replacing them with the sub-tree of the module. If this is the only module instance in the floorplan, then disconnecting one module alternative would leave any existing module alternatives floating around without parents, and they would be lost. To prevent this, a copy of the parent module is made if only one instance node exists. Jumping slightly ahead, refer to Figure 4.10. If instance A^i in module M were the only instance of module A in the entire tree, then a copy is made of module M. With two instances of module M, disconnecting an instance A^i and a single module alternative of A would still leave the remaining alternatives of A attached to the other instance in the copy of M. If M had no alternatives then not creating a copy of M would eliminate the possibility of floorplanning M with other alternatives of A, so a copy of M is also made if M has no alternatives.

Deleting a single alternative of A in the expansion would also remove this alternative from other instances of this module throughout the tree. Since this is undesirable, a copy of the module being flattened is also made (in this example, a copy of A). Now the flattening can be done on the copy of the parent and the copy of the module alternative without affecting other instances in the hierarchy. One final consideration: in our example the instances C^i and D^i of module A have a distinct path to the top of the tree which allows C^i and D^i to be uniquely identified. Preserving this path will prevent confusing them with any similar instances in the parent module M. By failing to preserve these paths after the flattening, these instances would be missing an important link in their heritage. To see this in our example, flatten the instances A^i and B^i of module M, both of which contain an instance C^i. After the flattening the C^i instances would be indistinguishable from each other (Figure 4.10d).

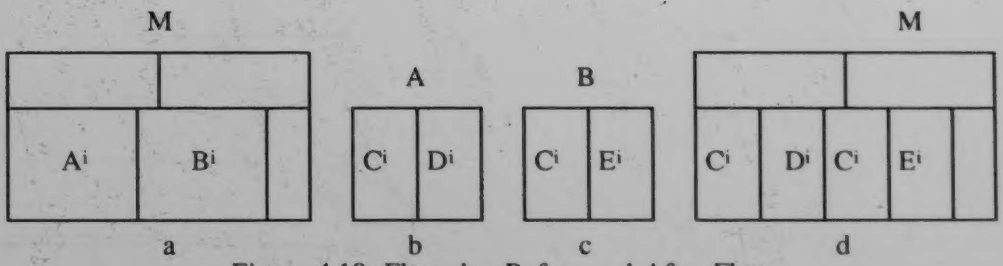


Figure 4.10: Floorplan Before and After Flatten

Appending the parent modules name $\boldsymbol{M}$ and instance name $\boldsymbol{A}^i$ to each instances schematic name before the flattening will eliminate the confusion. The schematic name for every instance starts with its module name appended in the field sch_name, as in $\boldsymbol{C}^i$:$\boldsymbol{A}$. Thus the sch_name of $\boldsymbol{C}^i$ which came from $\boldsymbol{A}^i$ changes from $\boldsymbol{C}^i$:$\boldsymbol{A}$ to $\boldsymbol{C}^i$:$\boldsymbol{A}/\boldsymbol{A}^i$:$\boldsymbol{M}$ and the sch_name of $\boldsymbol{C}^i$ which came from $\boldsymbol{B}^i$ changes from $\boldsymbol{C}^i$:$\boldsymbol{B}$ to $\boldsymbol{C}^i$:$\boldsymbol{B}/\boldsymbol{B}^i$:$\boldsymbol{M}$. This is still insufficient to guarantee distinguishability. Moving $\boldsymbol{D}^i$ into $\boldsymbol{C}^i$:$\boldsymbol{A}/\boldsymbol{A}^i$:$\boldsymbol{M}$ using cut and paste and then moving it back again as a component of $\boldsymbol{M}$ changes it's name to $\boldsymbol{C}^i$:$\boldsymbol{A}/\boldsymbol{A}^i$:$\boldsymbol{M}$:$\boldsymbol{D}^i$:$\boldsymbol{M}$, which does not reflect its true heritage. To address this concern paths are not appended when moving modules down in the hierarchy. Paths are only appended when moving up in the hierarchy if the module name at the end of the sch_name differs from the name of the module it is moving from, or differs from the name of the module being flattened. This does guarantee distinguishability. The following, *Update_Strings*, perform this. Using the above example, parent is a pointer to the record $\boldsymbol{M}$. Parent_path is the sch_name of $\boldsymbol{M}$. Node_name is the sch_name of $\boldsymbol{A}$.

```
Update_Strings (parent, parent_path, node_name)

BEGIN
child and c_node point to the first child of parent;
```

```
DO
    BEGIN
    IF (c_node is a module node)
        BEGIN
        IF (the sch_name of c_node is the same as node_name)
            Append path_name to c_node's sch_name;
        END
    IF (c_node is not a module node or we did just append)
        Update_Strings(c_node, parent_path, node_name);
    c_node points to c_nodes sister;
    END
WHILE (c_node is not the same as child)
END
```

Note that this procedure acts on all the instances of a module, recursively checking instances so long as they are still traceable to the module with the ***sch_name*** of ***parent_name***. So, moving instance A^i of module of M down two levels, and then M flattened to its parent N, will correctly update A^i's path to $A^i{:}M/M^i{:}N$ instead of to $A^i{:}M$.

The following procedure interleaves two iterated structures. Two things must be true before this can happen. Both records must have the same number of iterations and both must be children of the same node.

```
Interleave (inter0, inter1)
```

```
Check to make sure both structures are side by side and have the
                    same number of iterations;
Delete (inter1);
start0 points to child of inter0;
start1 points to child of inter1;
Update_Strings(inter1,inter1_path,start1);   /* Update path indices */
master_list points to next sister of start0;    /* start0 is a slicing node,
                    not a list of children */
    /* The following interleaves the two lists, starting with start0.
                    We combine two lists into one by alternating
                    records, taking one record from one list then
                    one record from the other list etc. until one
                    of the lists runs out. The rest of the nodes in
                    the other list are appended to the master
                    list.*/
DO
    BEGIN
    Delete (start1);
    Insert(master_list,start1);
    master_list points to start1;
    start1 points to new child of inter1;
    END
WHILE (master_list does not point to start0 and master_list is not
                    NULL)
Split (start1);
Spadd(inter0,start1);
```

```
Clean_Up (inter0); /* Get rid of any redundant nodes */
END
```

After the criteria have been met *Interleave* starts the process of interleaving records. If we treat the lists headed by ***start0*** and ***start1*** as arrays and ***start0*** has m elements and ***start1*** has n elements with m<n, then the final list looks like [***start0***$_1$, ***start1***$_1$, ***start0***$_2$, ***start1***$_2$, ***start0***$_3$, ***start1***$_3$, ..., ***start0***$_m$, ***start1***$_m$, ***start1***$_{m+1}$, ***start1***$_{m+2}$, ..., ***start1***$_n$] are. The final call to *Clean_Up* gets rid of any redundant slice nodes.

The operation most frequently used during floorplanning is the ***change_slice_direction*** operation. It is this operation which allows the designer to change the floorplan of Figure 4.11a into the floorplan of Figure 4.11c or back again. Before describing the algorithm for ***change_slice_direction***, some higher level discussion of the operation should be given. It is important to understand how this operation will affect a floorplan since undoing a change slice operation will often involve more than one step. In Figure 4.11a the slice which changed direction was the slice between ***D*** and ***E***. The parent to these instances also has module ***F*** as a child, along with ***D*** and ***E***. After the completion of the operation, ***E*** and ***F*** are still sisters, separated vertically from ***D***, rather than horizontally. Visually the operation appears to affect only the block consisting of ***D***, ***E***, and ***F***. This is always the case with ***change_slice_direction***. Only the block (i.e. ***D***, ***E***, and ***F***) which contains the slice whose direction is being changed (between ***D*** and ***E***) appears affected while blocks outside this range appear unaffected. After this operation is complete, the block structure of the module record (or the parent record of the slice which changed direction) is rearranged, so that the operation cannot be reversed with a single operation.

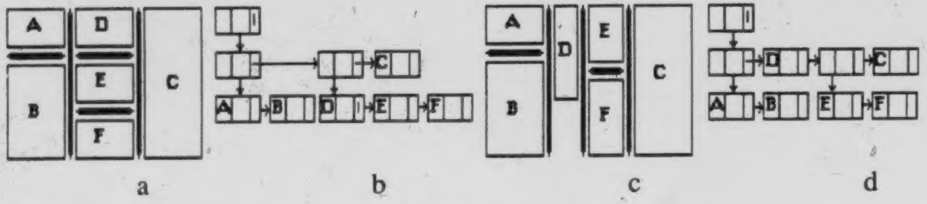


Figure 4.11: Changing Slice Direction and Tree

Several slice directions may need to be changed to restore the floorplan to its original form. In Figure 4.11c the blocks of the module are (***A*** ,***B***), ***D***, (***E***,***F***), and ***C***. This is also indicated by the rearrangement of the trees of Figure 4.11b and 11d. Changing the direction of the slice separating ***D*** from ***E***, and ***F*** again would not restore the floorplan to that of Figure 4.11a, but rather to that of Figure 4.12a.

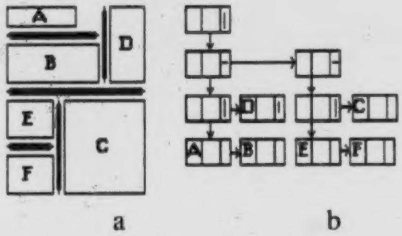


Figure 4.12: Floorplan and Tree After New Slice Direction Change

As a further example consider the floorplan of Figure 4.13a. This floorplan consists of 3 blocks, (***A*** , ***B***), ((***E***, ***F***),(***D***, ***G***)), and ***C***. Note that the middle block consists of two sub-blocks, (***E***, ***F***),(***D***, ***G***). Changing the slice direction between these two sub-blocks affects only the block ((***E***, ***F***),(***D***, ***G***)), as shown in Figure 4.13c. After the operation the module consists of 6 blocks, but two of the original blocks, (***A*** , ***B***) and ***C*** are unchanged. In example 13 the instances ***E***, ***F***, ***D***, and ***G*** become sisters of ***C*** and also become individual blocks of the module.

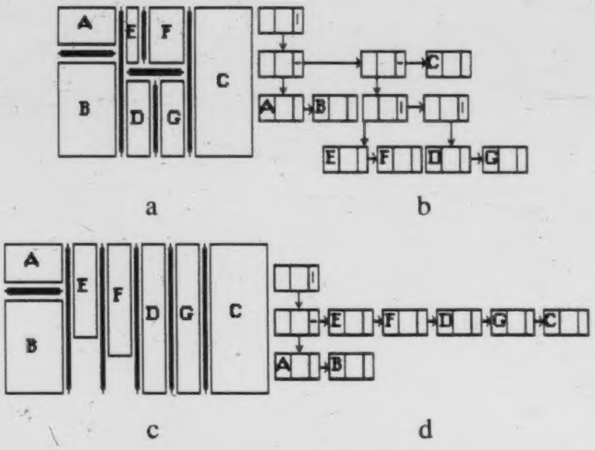


Figure 4.13: Changing a Slice Between Blocks

The example of Figure 4.14a shows a module which consists of three blocks, (***A*** , ***B***), (***D***, ***E***), and ***C***. If we change the slice direction separating (***A*** , ***B***) and (***D***, ***E***) then the floorplan will change to look like that shown in Figure 4.14c. In this example the module record has its slice direction changed from vertical to horizontal and the blocks of the module are now ***A*** , ***B*** and ((***D***, ***E***), ***C***).

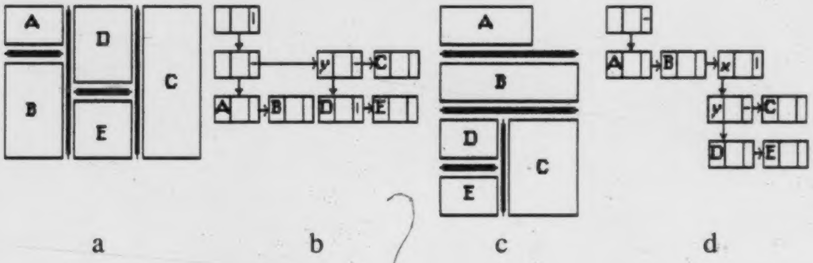


Figure 4.14: Changing Slice Separating Two Components

In this example the module record has its slice direction changed because the slice being changed separated its children (rather than its grandchildren as in the previous examples).In the previous examples effects on the parent of the slice being changed may not be obvious because the procedure *Clean_Up* removes redundant nodes after the operation is complete.

Changing the slice directions of iterated structures may have different effects on the floorplan than changing other slices. If the slice chosen to change direction is between iterations, then the iterations themselves will be separated by a different direction, as shown in Figure 4.15. The slice chosen in Figure 4.15a is between $\boldsymbol{D}_1$ and $\boldsymbol{A}_2$, with the resulting layout as in Figure 4.15c.

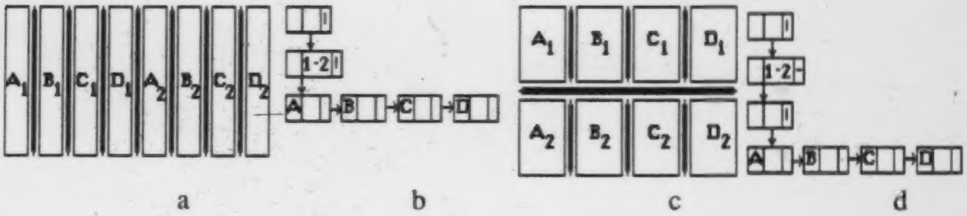


Figure 4.15: Changing Slice Between Iterations

Note that the iterations are separated vertically after the operation, but that the components of each iteration are still separated horizontally. Contrast this to the floorplan of Figure 4.16a, where the slice chosen to change direction was between $\boldsymbol{B}_1$ and $\boldsymbol{C}_1$. Note that changing the floorplan of the first iteration also affected the second iteration. Changing a slice direction within an iterated structure will affect all iterations identically.

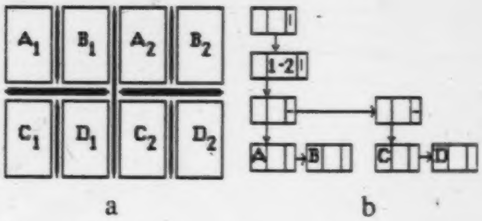


Figure 4.16: Changing Slice Which Separates Blocks of One Iteration

The procedure for changing slice directions is very simple.

```
identify slice (to be changed) which separates blocks as slice
identify parent node of blocks as parent_node

Change_Slice_Direction(parent_node, slice)
BEGIN
        If (parent_node is an iterated node)
                BEGIN       /* note that as long as the node identified is
                            iterated, this If code block will change slices
                            like that of Figure 4.15a to that of
                            Figure 4.15c or vice versa, because of the
                            clean up at the end of the If code block */

                add slice_node with same slice direction as parent node
                            between parent_node and parent_nodes
                            children (or child);
                change slice direction of parent_node;
                Clean_Up (parent_node);
                Return();
                END
        Endif
        slice_node is the node to the right of slice or below slice
                            (depending on slice direction);
        prev_node  is the node to the left of of slice or above slice
```

```
                    (depending on slice direction);
        Split list of children of parent_node into two lists, between
                    prev_node and slice_node;
        Provide each list with a new parent whose slice direction is the
                    same as parent_node. Call them new_par1
                    and new_par2;
        join new_par1 and new_par2 into a list and make this list the
                    children of parent_node;
        Change slice_direction of parent_node;
        If (parent_node is a module node)
            Clean_Up (parent_node);
        Else
            Clean_Up (parent of parent_node);
        Endif
    End Procedure;
```

Although the function of the procedure *Clean_Up* was presented earlier, it may be instructive at this point to discuss it again using the example of Figure 4.13. Figure 4.17 shows the progression of the hierarchy tree for this module as the slice direction is changed. Figure 4.17 shows the tree as it appears before the operation begins and Figure 4.17b shows the tree as it would appear in the procedure ***change_slice_direction*** before *Clean_Up* is performed. The list split in ***change_slice_direction*** had consisted of the two slice nodes with vertical slices shown in Figure 4.17a (one is the parent of ***E*** and ***F***, the other is the parent of ***D*** and ***G***). The parents added to them are the nodes marked ***x*** and ***y*** in Figure 4.17b. The *parent_node* is now the parent of nodes ***x*** and ***y***. Note that in Figure 4.17b *parent_node*'s slice direction has changed relative to Figure 4.17a. *Clean_Up* is called with the tree in the state shown in Figure 4.17b, and recursively searches the tree in a depth first manner looking for redundant nodes. The first redundant node it encounters is node ***x***. Node ***x*** is redundant because it has only a single child, so the horizontal slice which it represents does not separate two or more blocks. Therefore node ***x*** is removed and replaced by its children. The same argument applies to node y of Figure 4.17b, so it is also removed from the tree, leaving the tree in the state shown in Figure 4.17c. *Clean_Up* then identifies the parent node of ***E*** and ***F*** as being redundant since its slice direction is the same as its parent (the original *parent_node* whose slice direction has since been changed), so it is removed from the tree and replaced by its children.

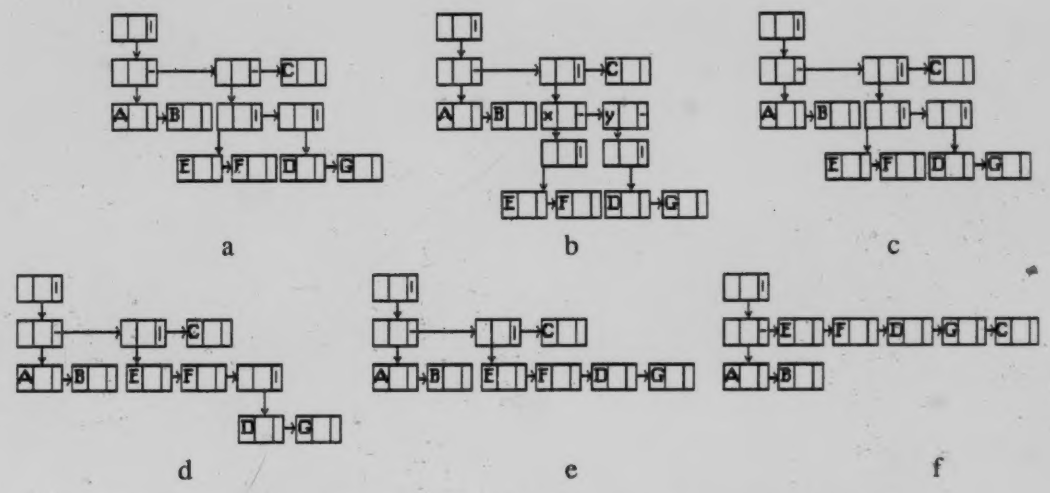


Figure 4.17: Steps of Clean Up After a Slice Direction Change

At this stage the tree would resemble Figure 4.17d. The same argument is applied to the parent of nodes ***D*** and ***G*** that was applied to the parent of ***E*** and ***F***, resulting in the tree of Figure 4.17e. Finally the original *parent_node* is deemed redundant by *Clean_Up* because it has the same slice direction as the module node at the top of the tree, so it is removed and its children inserted in its place. This results in the tree of Figure 4.17f, which is the final configuration. Although in this case the addition of nodes *x* and *y* appear unnecessary, there are some circumstances in which they will not be removed (the node marked *x* in Figure 4.14d is an example) and must be present to guarantee a working algorithm.

In the actual implementation several classes of slice direction changes are identified and treated independently in order to avoid unnecessarily adding and deleting nodes, but it could be replaced by the above procedure.

Cut_and_paste is the most difficult feature to implement. If the ***cut_and_paste*** does not move across levels of the hierarchy, or move from one iterated structure to another, then the straightforward procedure is to remove one node and its children and reinsert it somewhere else in the same modules sub-tree. This process may require some addition or deletion of nodes. More steps must be taken if the operation involves moving instances from one iterated structure to another. Assume, for the moment, that this operation moves a block from one multi-dimensional structure to another, but all within a single module. Before ***cut_and_paste*** begins, a comparison must ensure the iteration dimensions of both structures involved are identical. Only then may the ***cut_and_paste*** proceed, removing the node to be cut, with its sub-tree from its linked list, and inserting it into another linked list, next to the chosen paste node.

A ***cut_and_paste*** from one module to another requires more steps to be taken. First find the paths from the cut location to the top of the tree and from the paste location to the top of the tree and compare them from the root down. The last module record found which is common to both paths, the common module, is the node at which we will create a new alternative with the results of the operation. Next make a copy of each module which lies along the path from the cut location to the common module and from the paste location to the common module. Then remove the copy of the cut block and insert it next to the paste block. The following is the procedure for the ***cut_and_paste*** operation.

```
Cut_Paste_Copy (cut_node, paste_node, common_node)
        /*  cut_node is the record which, along with its children is to be moved.
            paste_node is the record which cut_node will be inserted next to.
            common_node is the lowest module record in the tree which has both cut_node and paste_node as its progeny */
BEGIN

        Make a copy of all the modules which lie along the path between cut_node and common_node (not including common_node). Assign a pointer to this new copy to temp_cut;
        Make a copy of all the modules which lie along the path between paste_node and common_node (not including common_node). Assign a pointer to this new copy to temp_paste;
        Make a copy of common_node call it new_common_node;
        Attach temp_cut to the instance in new_common_node which
```

```
                    is along its path.
        Attach paste_node to the instance in new_common_node
                    along its path.

          /* At this point cut_node has not been removed from the
             tree. All instances in new_common_node except the
             instance along the cut_path and the instance along
             the paste_path point to modules which existed in the
             tree before this procedure began. The instance along
             cut_path points to a new module, as does the
             instance along paste_path. The same can be said for
             all new modules along cut_path and all new modules
             along paste path. These new copies will no longer be
             dealt with, and will exist as an alternative to the new
             arrangement. From now on we deal with the original
             tree.*/
        Detach cut_node from its original location.
        Update sch_names of instance nodes below cut_node to
                    reflect new location in tree;
        Attach cut_node next to paste_node;
        Clean_Up (cut_nodes original parent module);
        Clean_Up (paste_nodes parent module);
    END;
```

Cut_and_paste updates the sch_names of all instances which are children of ***cut_node*** to reflect the full path from ***cut_node*** up to ***common_node***. Blocks can be cut and pasted from iterated structures in different modules by adding the steps mentioned for cutting and pasting across iterated structures to this procedure.

One of the features of FLINT is that it chooses the alternatives to use, so that it can minimize the area consumed in the layout, subject to aspect ratio constraints. This is done optimally by using an algorithm similar to Stockmeyer's algorithm [Stock83] as discussed in Chapter 2. For a given module layout, where each instance has several possible shapes, we apply Stockmeyer's algorithm directly to derive a set of shape boundaries for the module layout. The modification to this algorithm comes when combining boundaries of alternative module layouts. Where Stockmeyer combines two boundary lists into one by choosing (for blocks separated by horizontal slices) the maximum width of two blocks, and adding heights together, we merge the alternative boundary lists into a single list. We only remove from the list those layouts whose dimensions are larger than those of another layout in both height and width. The task is simple since we start with a sorted list which increases in height and decreases in width. The algorithm combines lists two at a time, adding to the composite list the next element from the two choices which has the smallest height, provided its width is less than that of the last element added to the list (note that its height must be greater, or the other element would already have been added). If the height of the two candidates are equal, then the one with the smaller width is chosen and the other discarded. The algorithm follows.

```
Procedure Merge (($h^U$,$w^U$), ($h^V$,$w^V$))   /* (h,w) is an array of heights
                                                    and widths, for blocks U and V respectively
                                                    */
BEGIN
```

```
Initialize $i = 1, j = 1$;
WHILE ($i < k$ and $j < n$)DO
    BEGIN
    add Merge_Join(($h^u_i, w^u_i$), ($h^v_j, w^v_j$)) to the list with pointers to ($h^u_i, w^u_i$) or ($h^v_j, w^v_j$);
    IF ($h^u_i \geq h^v_j$ and $w^u_i \geq w^v_j$) or ($h^u_i \leq h^v_j$ and $w^u_i \leq w^v_j$) THEN increment $i$ and $j$;
    ELSE IF ($h^u_i > h^v_j$) THEN increment $i$;
    ELSE increment $j$; /* because $h^u_i < h^v_j$ */
    END;
END;

Procedure Merge_Join(($h^u_i, w^u_i$), ($h^v_j, w^v_j$))
IF ($h^u_i \geq h^v_j$ and $w^u_i \geq w^v_j$) THEN ($x, y$) = ($h^v_j, w^v_j$);
ELSE IF ($h^u_i \leq h^v_j$ and $w^u_i \leq w^v_j$) THEN ($x, y$) = ($h^u_i, w^u_i$);
ELSE IF ($h^u_i > h^v_j$) THEN ($x, y$) = ($h^v_j, w^v_j$);
ELSE IF ($h^u_i < h^v_j$) THEN ($x, y$) = ($h^u_i, w^u_i$);
return ($x, y$);
END;
```

The algorithm for combining boundary lists is as given in Chapter 2. Since nodes may have more than two children, the algorithm must combine bounding curves for all children. The easiest way to do this, although not the most efficient, is to merge lists of boundaries for two nodes into a master list, and then merge the master list with lists for the rest of the children, one at a time. For iterated nodes the boundary heights or widths are multiplied by the number of iterations, depending on the slice direction separating iterations, giving the bounding curve for the whole structure.

After performing any operation on the tree, FLINT must derive a new set of bounding curves so that the change may be evaluated. The vast majority of modules are unaffected by changes to a single module, since changes cannot affect a module unless those modules have an instance of the altered module somewhere below it in the hierarchy. This provides us with a simple way to mark the nodes which need to have their boundaries recomputed. Assuming the change has affected all the blocks within the module modified, mark all internal nodes (but not instances) and then, starting at the module node climb the tree marking nodes with the following procedure. This procedure assumes that *node* has already been marked.

```
Mark_Nodes (node)
BEGIN
parent points to the parent of node;
IF (parent is not NULL and is not marked) THEN
BEGIN
    IF (parent is not an instance node) THEN
        BEGIN
        mark parent;
        Mark_Nodes(parent);
        END
    ELSE /* parent is an unmarked instance node */
        BEGIN
        temp points to next brother of parent in instance list;
        DO
```

```
                BEGIN
                mark temp
                Mark_Nodes(temp);
                temp points to next brother in instance list;
                END
            WHILE (temp ≠ parent)
        END
    END
```

This procedure marks nodes up to the root node (which has no parent), and marks all instance brothers of every module which has at least one node whose boundaries need to be recalculated (since all brothers may point to the modules whose boundaries have changed). If ***parent*** was marked previously then ***parent*** is an instance node and its brother instances were also marked previously, so no further action needs to be taken.

After marking all the nodes with this procedure, FLINT can recompute the boundaries for the whole layout using the modified Stockmeyer algorithm in a depth first traversal of the marked nodes. FLINT does not need to recompute boundaries for unmarked nodes and can use them verbatim in applying Stockmeyer's algorithm.

To mark boundaries when cutting and pasting between modules, FLINT must do two upward traversals, one from the module where the cut occurred and another from the module where it was pasted. The boundary recalculation can still be done with the downward tree traversal.

4.3 **Tools**

The floorplanning package makes considerable use of three tools which were available during development. The flooplanner uses Organized C to create and manage data structures, SunView for the mouse input and graphic output routines, and CapFast to create circuit schematics and parts lists.

4.3.1 Organized C

Organized C is a collection of macros, and processing routines, which support the creation and management of data structures. It is extensible insofar as the programmer can use it to create new data structures and macros for their management. The programmer can also use it to create new macros which operate directly on existing data structures. As an example structure, consider the trees produced by Organized C. Its construct is a parent with one pointer to a dually linked list of children, each with a pointer back to the parent. Each child may be the parent of another linked list. Macros for the tree structure are available to add new children to a node, insert new sisters in a list, delete records from a linked list (with its list of children intact), find the parent of a record, the forward or backward sister of a record, or the child of a record. There are also macros for traversing a list of records and creating new records. The macros and routines only deal with record pointers. The programmer uses Organized C's routines by adding a macro definition to an otherwise normal record definition. The macro defines all the pointers which Organized C uses to link records together.

The floorplanning hierarchy tree is a three dimensional tree, constructed specifically for this application. Besides the linked lists of sisters which exist in the standard tree, the three dimensional tree adds a second dually linked list of brothers to each record (hence three dimensions instead of two). Unlike sisters, brothers may have different parents. FLINT only uses brothers in instance nodes which point to a linked list of similar instances throughout the tree. Their use has been outlined in the description of data structures near the beginning of this chapter.

FLINT uses several newly created macros to operate on the hierarchy tree, in addition to those available for binary trees. These new macros can disconnect a parent from its child, connect a parent to a new child, split a list into two lists, and add a list of children to a parent at the end of its existing list. These macros do nothing that cannot be done with the existing macros, but require fewer operations and are more convenient to use. Some of these operations would adversely affect the tree if performed out of context. For example, connecting a parent to a child disconnects the parent from its existing list of children, and disconnects the child from its existing parent. This macro is only invoked after a *Copy_Tree* operation, and only on the instances within the new module copy, when its child will be a module node with two or more parents. In an instance like this, no nodes are lost. Since *Copy_Tree* just allocated the newly connected child it has no parent.

4.3.2 SunView Graphics

The graphics routines were written using the SunView Graphics system. The program uses this package in functions which draw graphics, open and manage windows, and for input routines using the mouse. SunView allows FLINT to work with vector graphic commands on an artificial canvas whose size FLINT determines. The program may draw on any part of the canvas whether that part of the canvas is displayed or not. The view of the canvas takes place through a scrollable and resizable window. SunView responds to user input to control which part of the canvas is in view. The user can adjust the window as he would any other window in the Sun environment. For this floorplanner, the canvas and window sizes of the layout display are the same, so that the user views the whole canvas in the window. If the user resizes the window, then the canvas shrinks or expands to fully fill the window. From the programs point of view, the canvas has the same dimensions before and after resizing the window.

Two other windows are also opened by FLINT via SunView, one for statistics, and one to display the hierarchy tree of the module displayed in the layout window. The latter of these windows is resizable and smaller than its underlying canvas and uses scroll gadgets to change the display. Mouse input is also controlled using SunView By using SunView the program can determine the mouse location within the window and detect mouse button depressions. SunView also supports the display of menus and the selection of menu items. Depressing the rightmost mouse button activates menu display. Moving the mouse will highlight various menu items and releasing the mouse button will select the highlighted option. This is the same procedure that is used to select menu items with most Sun software.

4.3.2 CapFast Schematic Editor

FLINT's program uses the CapFast circuit and schematic layout package to define schematic diagrams of circuits and to obtain circuit part lists. This package has a full set of standard gate and device symbols and allows the design of custom symbols for custom modules. Using existing standard symbols and custom defined symbols the user can create schematics hierarchically. FLINT uses the hierarchical schematic definitions to determine module hierarchy in the floorplan. A module is created by interactively connecting the terminals of symbols together. The components of a module are stored as symbols and terminals, not as other module definitions. Since each symbol is the name of a module which also has a definition in a file, this means that file definitions are hierarchical. To convert the schematic to a floorplan, a routine in FLINT reads a CapFast file and, using the symbols found in the current module definition, recursively reads all other CapFast files associated with those symbols. Symbols which have no CapFast schematic file are assumed to be atomic leaf cells. FLINT creates a part list of all the leaf cells it finds in this circuit and compares this against an inventory of existing standard cells and module cells. If at least one layout exists for each leaf cell, then FLINT creates an initial layout and the floorplanning can begin. The first time FLINT encounters a circuit it creates a floorplan file. This file is used subsequently, so parsing of the schematics only needs to be done the first time FLINT is run. Afterwards, if the floorplan file exists, FLINT does not need to read the CapFast files. The floorplanning file contains the floorplan definition, including alternative layouts and slice directions. This file is written back to disk after every operation is complete so that even a catastrophic breakdown will not destroy all the designers work.

Chapter Five: Results

This chapter presents some information about FLINT and the results obtained by floorplanning two circuits. The first section presents some statistics on FLINT. The second section describes the circuits that were floorplanned; the third section describes some principles applied by the author during the process, and the fourth section describes the results of the floorplans, including sizes, aspect ratios etc. The fifth section describes some impressions about FLINT and the sixth section discusses changes which we would make if we started the project over again. This last section does not discuss additions which might be made to FLINT, only implementation details which would change if it were rewritten.

5.1 Information About FLINT

FLINT was written in the C programming language. The source code contains roughly 4700 lines and 265,000 characters. The executable is about 350Kbytes, a figure which includes the C libraries and SunView executable code. Organized C incorporates macros into the source code, rather than the object modules, but undoubtedly adds to the size of the executable image.

5.2 Circuit Description

The example circuit floorplanned was an AM2901 bit slice processor originally designed by American Micro Devices (AMD). The AM2901 is a four bit wide processor slice containing an ALU, a 4x4 array of dual port RAM, several two and three bit MUXes, an instruction decode unit, an arithmetic function decode unit and some steering or glue logic. Appendix A shows a block diagram of the AM2901. This circuit was chosen as an example because it contains enough components to be a considered a realistic test for FLINT, it has several iterated components (such as the memory of the bit slice), there is some random logic in the circuit, and some modules exist throughout the circuit, allowing us to test the reuse of module floorplans.

The example circuit was floorplanned in three different invocations. The first test used a circuit hierarchy that extended deep enough that some modules were composed primarily of transistors (specifically the dual port RAM) while others were composed of elementary gates. Not all modules in the first test were described at this level of detail, some cells still performed complex functions. Each leaf cell was allowed to have up to four alternative dimensions. The hierarchy used in the second test did not extend as deeply as the first. Each lowest level leaf module contained at least one cell which performed a high level function (such as multiplexing two signals), although the leaf modules may still have contained gate level glue logic. All modules in the second circuit's hierarchy also appeared in the first circuit's hierarchy, but the hierarchy of the first circuit extended beyond the leaves of the second circuit. We included the second test because we felt that in practice, designers would not use this floorplanner at the transistor level; most modules would be available in a standard library.

The third test was used as a benchmark. In this test only one alternative was used for each module, although leaf cells still had the same range of alternative dimensions as in test one. The circuit hierarchy for this test was identical to the circuit hierarchy used for the first test. The alternative chosen for each module was identical to one of the alternatives used in test one, and was chosen with the expectation that it would be a good choice in producing a layout with a normal floorplanner (which did not support alternatives). The By comparing the results of this test with the results of the first test we hoped to determine the usefulness of including module floorplan alternatives in FLINT.

Both circuits have three atomic modules, representing high level functions, in the block diagram. These three blocks are the address destination decode (ADD), the address source operand decode (ASOD), and the arithmetic function decode (AFD). These blocks are responsible for converting the instruction bit patterns into the bit patterns required to perform the instructions. It was not felt that working out combinational logic to perform these functions would enhance the evaluation of FLINT, and so they were left as leaf cells. In the floorplan these cells could represent combinational logic constructed with standard cell logic gates, as a gate matrix, or using some other method. The sizes chosen for these blocks may appear small, but the instruction set AMD used was likely chosen to minimize the amount of logic needed to perform these functions, given that the other components were already available, and so these three blocks probably contained very few gates.

We did not have a comprehensive library of modules available to use in floorplanning. The Magic standard cell library available had descriptions of cells such as D flip flops, NAND and NOR gates, but no cells implementing higher level functions. Rather than designing modules with several layout dimensions, some estimation was done on the expected size of the modules and the floorplanner used the estimates rather than actual cell dimensions. When a leaf cell was available in the standard cell library, FLINT used its dimensions as one alternate layout. Any subsequent alternate dimensions for a cell were based on the area of the available alternative. Leaf cells taking advantage of this include most logic gates, such as NAND and NOR gates, the N and P transistor types, flip flops, etc. Low level leaf cells such as these were created with two alternative dimensions for FLINT to choose from. More than two alternate NAND gates (with the same drive capability) would be unrealistic in a cell library. Larger modules had four or fewer alternate dimensions to choose from.

5.3 Steps Involved in Circuit Design

The first step in creating the test circuit was to decide which modules would exist at the lowest level of the hierarchy. Most of these modules were obtained directly from the block diagram of the AM2901. Other modules, such as the dual port RAM, were described manually. We included other modules, such as the Instruction Decode, atomically, since the purpose was to floorplan a circuit, not design a working alternative to the AM2901.

The second step in the design was to create a schematic description of the circuit using the CapFast schematic design editor. The circuit was entered hierarchically with symbol and schematic definitions created in the process.

The next step in the design process was to create a parts list of the lowest level modules of the AM2901. This information was available from the schematic and symbol files created by CapFast. The designer created a size file by combining the part list file with sizes for each leaf cell. This size file is normally created by scanning the standard cell library but, as stated previously, the available library did not have the necessary parts. FLINT used the CapFast files, and the size file to generate an initial floorplan. The initial floorplan hierarchy directly mimicked the schematic hierarchy, but contained no slicing records, as noted in Chapter 4.

The author then modified the initial floorplan by roughly following these rules:

1. Change the view down to an unfloorplanned low-level module. Move instances around within the module next to other instances to which they are tightly connected, with some consideration of the external connections. From this initial floorplan, create two more floorplans with the create alternative operation. Rearrange one of the three floorplans to be tall and narrow, a second to be short and wide, and the third to be roughly square.
2. Move up a level in the hierarchy and choose one of the other module instances to floorplan and repeat Step 1.
3. Repeat Step 2 to floorplan all module instances in the same way, then start rearranging the parent module using the procedure of Step 1. Check the instance modules to see if any of them can be more optimally arranged, and create new alternatives for these arrangements.
4. If a module has too many instances to floorplan comfortably, divide the module up into several blocks of related components, create new levels in the hierarchy for each of them, and floorplan them separately. When done, destroy these additional levels and rearrange the module if necessary. Although these newly created levels could be left intact, once they were partially rearranged, the designer could optimize them by going back to the original module organization.
5. Apply the previous steps to floorplan the whole circuit.
6. Go back to low-level modules and rearrange any that appear could benefit

from rethinking their floorplans.

7. If the designer noted some arrangement of blocks with a significant amount of dead space, but not enough to justify moving an instance from another block into it, then he considered moving some components from an adjoining module into this block. Before doing this he created an alternative, so that he could restore the original if this proved to be a mistake.

This is, of course, a very rough guide on how circuit was floorplanned. Since the designer had limited previous floorplanning experience, these steps should not be thought of as a tutorial in circuit floorplanning.

5.4 Floorplan Results

When evaluating the success of a floorplanning utility, two important factors are the size of the floorplans which the floorplanner creates, and the ease of use. In our present situation there is no facility to route a floorplanned IC. The routers available were channel routers which are not suitable to this application. Therefore the only way to measure the performance of the floorplanner was by measuring the area wasted in the floorplan. We present the results of the two circuits, stating the minimum area consumed and the actual area consumed in the floorplan.

Although both circuits are floorplans of an AM2901 bit slice processor, the area which each consumes is different. In the second circuit the modules ramcolsens, trans, alubit, mux2, mux3 were atomic. The sizes created for these modules were smaller than the smallest sizes given in the floorplan of the first test. The created sizes were smaller since modules like these in a standard cell library would be more compactly placed than they appear in the floorplan of circuit one, since the designer would be trying to optimize their placements. No more than two aspect ratios were created for each of these modules. Other components were left with the same sizes as those of circuit one.

The sum of the minimum area of each component in circuit one was 7,817,640 CIF[5] units, while the actual area consumed for a reasonable floorplan was 9,033,780 CIF units. Components in this floorplan were arranged while keeping in mind probable external connection locations. By relaxing this requirement somewhat, the area consumption was reduced to 8,921,440 CIF units with little effort.

While floorplanning circuit two more consideration was given to a compact design than to terminal placement. The sum of component sizes in circuit two was 7,502,240 CIF units. The floorplan for this circuit differed from that of circuit one in several places, the final size being 8,708,000 CIF units.

The floorplan for the third circuit was created by running FLINT on the final floorplan of circuit one, and removing all but one alternative for each module. At each step the floorplan was re-examined to make sure that the remaining alternative would be a "reasonable choice for a designer to make", but the alternatives were not custom made to fit into the final floorplan. The floorplan created by FLINT for this test used 10,247,520 CIF units. This is 15% larger than the area consumed in test circuit one, and demonstrates the usefulness of using module floorplan alternatives. Output of the final floorplans for tests one, two and three are included in Appendix B. These floorplans are not drawn to the same scale, however, comparing the floorplan from test one to that of test three shows the improvement in area which can be achieved by using alternative module placements. Appendix C shows a screen image of the final floorplan from the first test.

---

[5] a scalable unit of linear measurement usually equal to a hundredth of a micron: in this context they refer to area and should actually be CIF units^2

5.5 Performance of FLINT

5.5.1 Impressions About FLINT

As expected, it took some time to become familiar with the floorplanner. The function which took the most time to become comfortable with was changing slice directions. While learning how to use FLINT, changing slice directions would not seem to behave as expected, although the floorplanner was doing exactly what it was supposed to do. After spending some time with it, I learned how to organize the use of functions to achieve the desired effect.

The operation of most other functions was predictable either the first time used or after a very few uses. The inability of cut and paste to paste at the end of a row of blocks was annoying, but was easily corrected with the change order of children operation.

I was surprised at how quickly the program recalculated floorplans boundaries and redisplayed the screen. I had expected a noticeable delay between the time a change was made to the floorplan and the redisplay of the new floorplan.

The statistics provided were meant to allow the designer to compare how a change affects a floorplan, but in most circumstances rearranging a floorplan required several operations. The statistics changed with each operation, making comparisons difficult. The work around to this was to create an alternative floorplan, make the proposed changes to the new copy, and then compare the statistics of the new design with those of the old design. While this is a little clumsy, there are no alternatives (which I can think of) which require fewer step on the part of the designer to do a comparison.

5.5.2 Suggested Changes to FLINT

If FLINT were rewritten, with our new knowledge, there are some changes which would be made. This should not be confused with enhancements or additions which may still result from future work, but are implementation details, all concerning designer interaction with the floorplanner which the developer would change.

The method of changing slice directions could be improved. Rather than changing a slice direction, which may affect the entire circuit, the designer should be able to specify a block or several blocks which the operation will affect, while blocks outside this range would remain unaffected. For example, consider changing the floorplan of Figure 5.1a to that of Figure 5.1b. Normally this would require several operations.

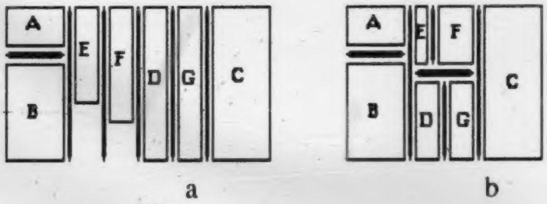


Figure 5.1: Change Slice Direction

If the designer could select just blocks ***E***, ***F***, ***D***, and ***G***, and then change the direction of the slice separating ***F*** from ***D***, the floorplan would change to that of Figure 5.1b in one step.

The inability to paste blocks at the end of a row is annoying. A menu option to allow pasting behind the selected paste block as opposed to ahead of it could solve this problem.

The view only shows one module at a time. The ability to show several modules around it, without the need to expand a level and recreate it afterwards would be a benefit.

FLINT could also benefit by the addition of an undo command. There were times during the design phase where several changes were made which I wished to undo. The only recourse here was to abandon the whole session, and restart with a backup copy. Fortunately FLINT creates a backup copy at the beginning of every session.

Overall, the performance of FLINT is satisfactory. Most operations work in a straightforward manner, requiring very little experience to get used to. This does not refer to experience in floorplanning itself, just the time it takes to get used to the tools. Updates are very fast (no noticeable delay) and rearranging the floorplan of a module can be done with relative ease.

Chapter Six: Conclusions

This chapter presents some conclusions about FLINT. The first part of this chapter presents a synopsis of the thesis. The second part discusses the success of the effort, and how well FLINT has met its goals. The third part discusses the acceptability of slicing structures. The fourth part presents some limitations of FLINT, and the final section presents some ideas for future research.

6.1 Synopsis

After having reviewed previous work in floorplanning, we felt there was room for improvement. Most current floorplanners proceed either by following algorithms or by using heuristics, while others use knowledge bases to create designs. One thing all these floorplanners have in common is that they either create a single floorplan, without checking other possibilities, or they give intermediate designs little consideration. Another feature of these floorplanners is that the human designer is left out of the design process. Although this saves labour, it also fails to take advantage of the human designers adaptability in solving problems. Where a human can foresee a problem with a procedure and take steps to correct it, the algorithmic and heuristic algorithms often cannot. Most floorplanners fail to take advantage of a multi-level hierarchy that can reduce the size of the problem by decreasing the number of components under consideration at any time. This, in turn, reduces the solution space, and allows for a more concentrated effort on the portion of the problem under consideration at any time.

We wished to create a floorplanning utility which could improve on the existing solutions. Our goals were to create an interactive floorplanning utility that was simple to use, but still capable of allowing the designer to explore a wide range of possible floorplan solutions. To this end we created a floorplanning utility in which multiple module floorplans may be designed and considered for inclusion in the final floorplan, only removing them from consideration when the existence of a better solution is known. The human designer plays an important role in the floorplanning process. He may address the floorplanning problems he feels are most pertinent, addressing them in the order he wishes. FLINT employs a multi-level hierarchy which follows the hierarchy of the schematic so that the designer need not bite off more than he can chew. Even this hierarchy can be modified if the designer feels it ought to be. FLINT uses slicing structures which allow hierarchical floorplans to be represented easily, and allows fast recalculation of the statistics.

FLINT includes several operations which are easy to use and allow any conceivable slicing floorplan to be created. Feedback on the results of a change is provided after each operation. A graphical user interface allows the designer to see what the results will look like and he can choose his next action accordingly.

6.2 Success of FLINT

We believe we have met most of our objectives. Most of the operations are straightforward and easy to use. Some of them, ***change_slice_direction*** and ***cut_and_paste*** in particular, are not as straightforward, or as predictable as desired, but are still workable. Use of the multi-level hierarchy was generally successful. The levels in the hierarchy are convenient points for creating alternatives, and they break the circuit down into smaller chunks that are easy to work with. However, when the designer is floorplanning a module, he does not get the global sense of the circuit we were hoping for. It is too easy to lose track of a modules external connections.

We feel that the use of alternatives was largely responsible for the small amount of wasted space in the design of the AM2901. Although floorplanning alternatives may appear to increase the amount of time the designer spends floorplanning a circuit, this time is probably recovered by eliminating the need to backtrack and floorplan the same modules several times. Using module definitions in multiple instances does allow FLINT to recalculate bounding curves very quickly on floorplans of realistic size.

The initial floorplan created by FLINT, where a single slice separates all the components of a module, is annoying to work with. Often the components are so small that FLINT cannot display text within the defining rectangle, making them difficult to identify, and thus difficult to floorplan. Components which will ultimately be close together may start off some distance apart.

Overall we are happy with the results of the project and believe that we have shown the usefulness of the concepts we were using. Floorplanning the AM2901has shown us that creating alternative module floorplans, and choosing among them can result in smaller floorplans. It has also shown that reusing module floorplans in multiple instances can significantly reduce the amount of computation involved in boundary calculations. Floorplan boundary updates were quick, and it does not take long to familiarize yourself with any of the operations. Floorplans can be created quickly and waste little space (about 15%).

6.3 Performance Using Slicing Structures

There was some question at the beginning of the development process about how well slicing structures would perform. There were questions concerning the limitations imposed on floorplans by slicing structures, and whether the loss of flexibility in floorplan design would be offset by the increased functionality of FLINT. Our example floorplan shows that the wasted space is about 15%. This is not very high and indicates that slicing structures are adequate to the task of floorplanning, imposing no more than a small penalty on the quality of the final floorplan while decreasing the complexity of the algorithms involved in solving the floorplanning problem.

6.4 Limitations of FLINT

Even if we corrected all the problems mentioned so far, FLINT would still have limitations. It is important to know what these limitations are, either so they can be addressed in the future, or so that the designer knows what external information he may need to seek.

FLINT does not have an interface to a leaf module generator. All modules must be available before floorplanning begins. FLINT should work with leaf modules described only by an estimated bounding box formula, rather than several rigid dimensions.

In its present state FLINT cannot create an acceptable floorplan without a human designer, nor even an initial floorplan as a starting point. Creating an initial solution (or set of solutions) that was closer to a final result would decrease the amount of time the designer needs to spend to complete a floorplan.

FLINT can display the components of only one module at a time. The result of this is that the designer has little appreciation of the whole circuit when designing at the lower levels of the hierarchy. As long as this is an interactive tool, the designer should be able to look at the entire circuit and zoom in on the module he is presently floorplanning.

The most important limitation of this floorplanner is the lack of routing information during floorplanning. The addition of wiring nets to a module increases the area it consumes considerably, and can change dramatically based on the module's floorplan. An accurate global estimation of channel density may provide much insight into space saving changes which may be made to a module's floorplan.

Another problem, related to the lack of routing information, is that the designer has no way of knowing which modules connect to which other modules, without referring to the schematic diagrams. Some method of showing the interconnections within a module and the external connections would help the designer choose floorplans which minimize routing space and wire length.

Despite the fact that our experience with FLINT suggests that slicing structures are adequate for floorplanning circuits, support for some form of non-slicing structure is desirable.

6.5 Future Work

Many of the limitations just mentioned can and should be changed in the future. These are not trivial additions and would require some effort. These changes are listed according to the amount of time it would take to implement them, with the quickest given first. This is not necessarily the order in which they should be added to FLINT.

The first addition to FLINT should concern initial floorplan creation. This requires very little modification of the existing program, but does require modest additions, and can provide an excellent starting position for interactive floorplan design. Using a good off-the-shelf algorithm, such as min-cut, would help reduce development time. Furthermore, at least three such placement algorithms have already been written at the University of Alberta, and could be converted for use by FLINT in a reasonably short space of time.

The second addition to FLINT should be support for a module generator. The preferred method of floorplan design is to design modules after floorplan completion rather than designing the floorplan using existing modules. This will allow the module generator to place external connections intelligently, reducing interconnect length. Adding this support requires replacing the boundary recalculation algorithm given by Stockmeyer [Stoc83] by an algorithm similar to the one developed by Otten [Otte83], which combines boundaries described by a set of curves rather than a set of points. Some method of adding these curves to the definition of a module must also be found.

The third addition to FLINT should be estimation for routing channel width, perhaps similar to the system used in Mason [LaDi85]. Although this addition would take some time, particularly since FLINT now ignores connections, it would substantially increase the accuracy of the on screen representation with which the designer works. Although it will slow the floorplan recalculation down to the point where there is a noticeable delay between operations, the benefits probably outweigh this cost. The impact of this delay could be reduced by performing the recalculation of boundaries only when the designer requests it, so he could make several modifications before recalculating routing area.

The fourth, and most ambitious, addition to FLINT should be an expert system to drive the floorplanning process since the ultimate goal of IC design tools is to completely automate the design process. Access to an automated floorplanner which creates high quality floorplans would remove a large barrier to custom IC design. One hurdle to overcome in this endeavor is that the U of A has no human floorplanning expert from which to model a set of expert rules. Furthermore, projects of this type done at other institutions have taken years to complete, and this would likely also be the case with a fully-automated expert floorplanner.

References

[WaTh85]
R. A. Walker and D. E. Thomas, "A Model of Design Representation and Synthesis" Proc. 22nd Design Automation Conference, pp. 453-459, 1985

[ScUl72]
D. M. Schuler and E. G. Ulrich, "Clustering and Linear Placement", Proc. 9th Design Automation Conference, pp. 50-56, 1972

[Breu77]
M. A. Breuer, "Min-Cut Placement", Design Automation and Fault-Tolerant Computing, Vol 1, No 4, pp. 343-362, October, 1977

[KeLi70]
B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", Bell Sys. Tech. J., vol. 49, pp. 291-308, February 1970

[ScKe72]
D. G. Schweikart and B. W. Kernighan, "A Proper Model for the Partitioning of Electrical Circuits", Proc. 9th Design Automation Conference, pp. 57-62, 1972

[FiMa82]
C. M. Fiduccia and R. M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions", Proc. 19th Design Automation Conference, pp. 175-181, 1982

[DuKe85]
A. E. Dunlop and B. W. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits", IEEE Transactions on Computer Aided Design, Vol CAD-4, No 1, pp. 92-98, January 1985

[Goto81]
S. Goto, "An Efficient Placement Algorithm for the Two-Dimensional Placement Problem in Electrical Circuit Layout", IEEE Transactions on Circuits and Systems, CAS-28, No. 1, pp. 12-18, January 1981

[PrVa80]
B. T. Preas and W. M. VanCleemput, "Placement for Arbitrarily Shaped Blocks", Proc. 17th Design Automation Conference, pp. 474-490, 1980

[Prea79]
B. T. Preas, "Placement and Routing Algorithms for Hierarchical Integrated Circuit Layout," Stanford Univ., Ph.D. Thesis, 1979

[Fell76]
Feller, "Automatic Layout of Low-Cost Quick-Turnaround Random-Logic Custom LSI Devices," Proc. 13th Design Automation Conference, pp. 17-85, 1976

[ShDu85]
L. Sha and R. W. Dutton, "An Analytical Algorithm for Placement of Arbitrarily Sized Rectangular Blocks", Proc. 22nd Design Automation Conference, pp. 602-608, 1985

[Stoc83]
L. Stockmeyer, "Optimal Orientations of Cells in Slicing Floorplan Designs", Information and Control, Vol. 57, pp. 91-101, 1983

[LaDi85]
D. LaPotin, S. Director, "Mason: A Global Floorplanning Tool", 1985 Custom Integrated Circuits Conference, pp. 143-145, 1985

[Otte83]
R. Otten, "Efficient Floorplan Optimization," International Conference on Computers and Design, pp. 499-502, 1983
[Moda88]
H. Modarres et al, "Floorplanning of Hierarchical Layout in ASIC Environment," IEEE Custom Integrated Circuits Conference, pp. 7.1.1-7.1.4, 1988
[WoLi86]
D. F. Wong and C. L. Liu, "A New Algorithm for Floorplan Design", Proc. 23rd Design Automation Conference, pp. 101-107, 1986
[Naha86]
S. Nahar et al, "Simulated Annealing and Combinatorial Optimization," Proc. 23rd Design Automation Conference, pp. 293-299, 1986
[Ueda85]
K. Ueda et al "CHAMP: Chip Floor Plan for Hierarchical VLSI Layout Design", IEEE Transactions on CAD, CAD-4 No. 1, 12-22, January 1983
[CaHe88]
H. Cai and J. Hegge, "Comparison of Floorplanning Algorithms for Full Custom ICs", IEEE Custom Integrated Circuits Conference, pp. 7.2.1-7.2.4, 1988
[WaAc87]
H. Watanabe & B. Ackland, "FLUTE An Expert Floorplanner for Full-Custom VLSI Design," IEEE Design & Test, pp. 32-41, 1987
[KoKi84]
K. Kozminski and E. Kinnen , "An Algorithm for Finding a Rectangular Dual of a Planar Graph for use in Area Planning for VLSI Integrated Circuits", Proc. 21st Design Automation Conference, pp. 655-656, 1984
[JaSk87]
M. A. Jabri and D. J. Skellern, "Implementation of a knowledge base for interpreting and driving integrated circuit floorplanning algorithms," Artificial Intelligence in Engineering, Vol. 2 No. 2, pp. 82-92, 1987

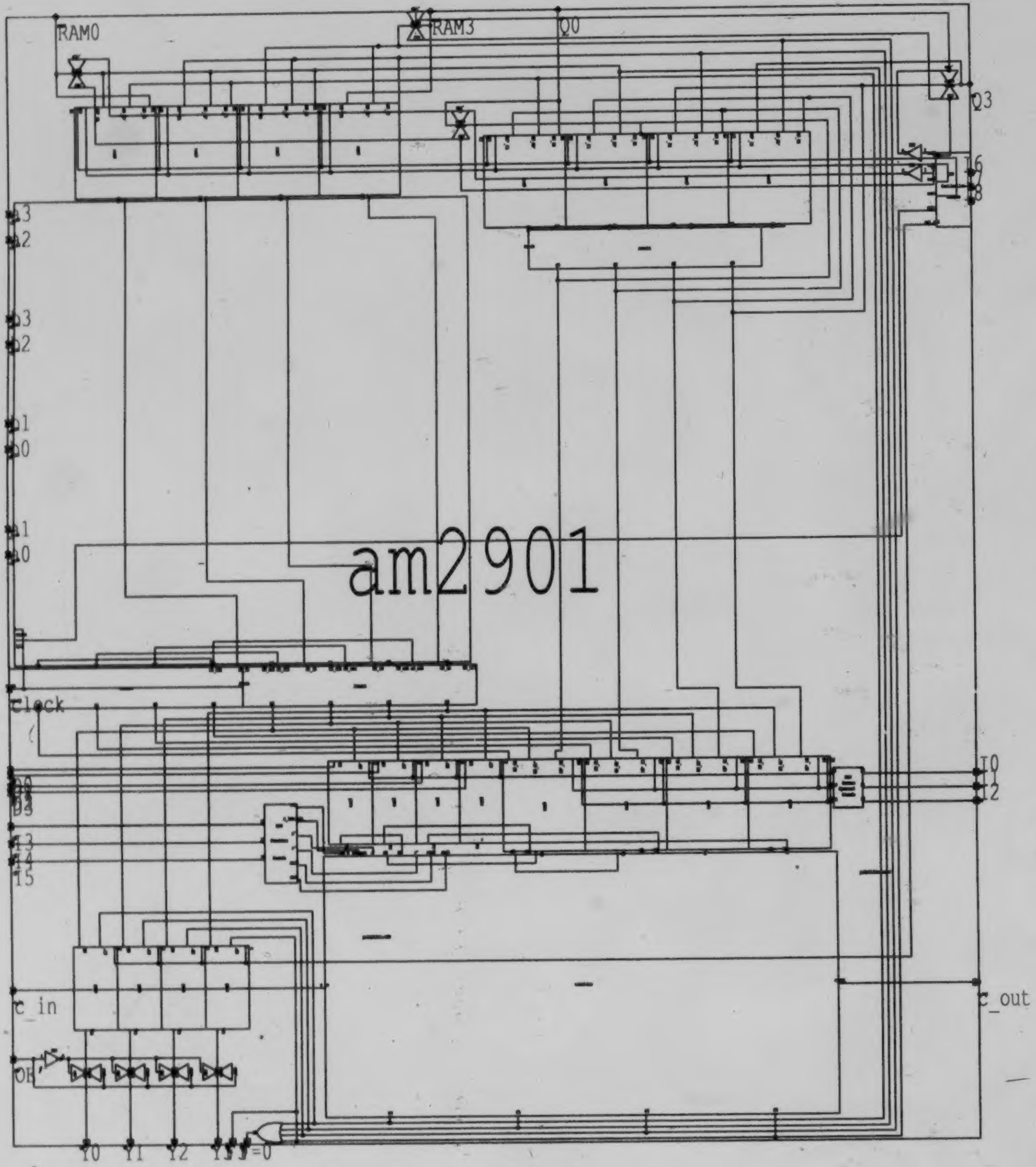

RAM0
RAM3
Q0
Q3
I6
I7
I8
A3
A2
B3
B2
B1
B0
A1
A0
am2901
clock
D0
D3
I3
I4
I5
c_in
OE
Y0
Y1
Y2
Y3
F=0
I0
I1
I2
c_out

Figure A.1: Block Diagram of AM2901, From Schematic

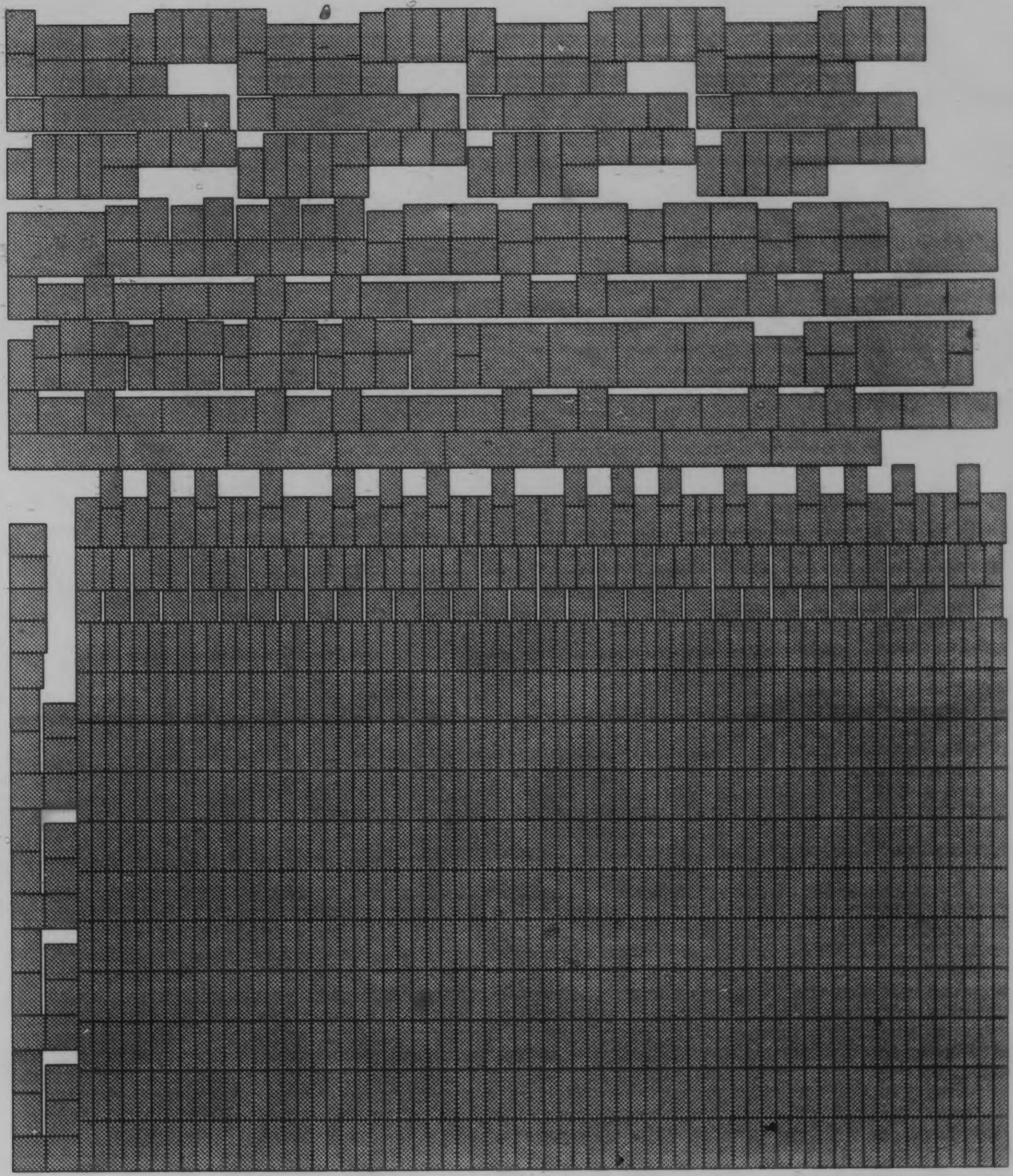


Figure B.1: Layout of Test One

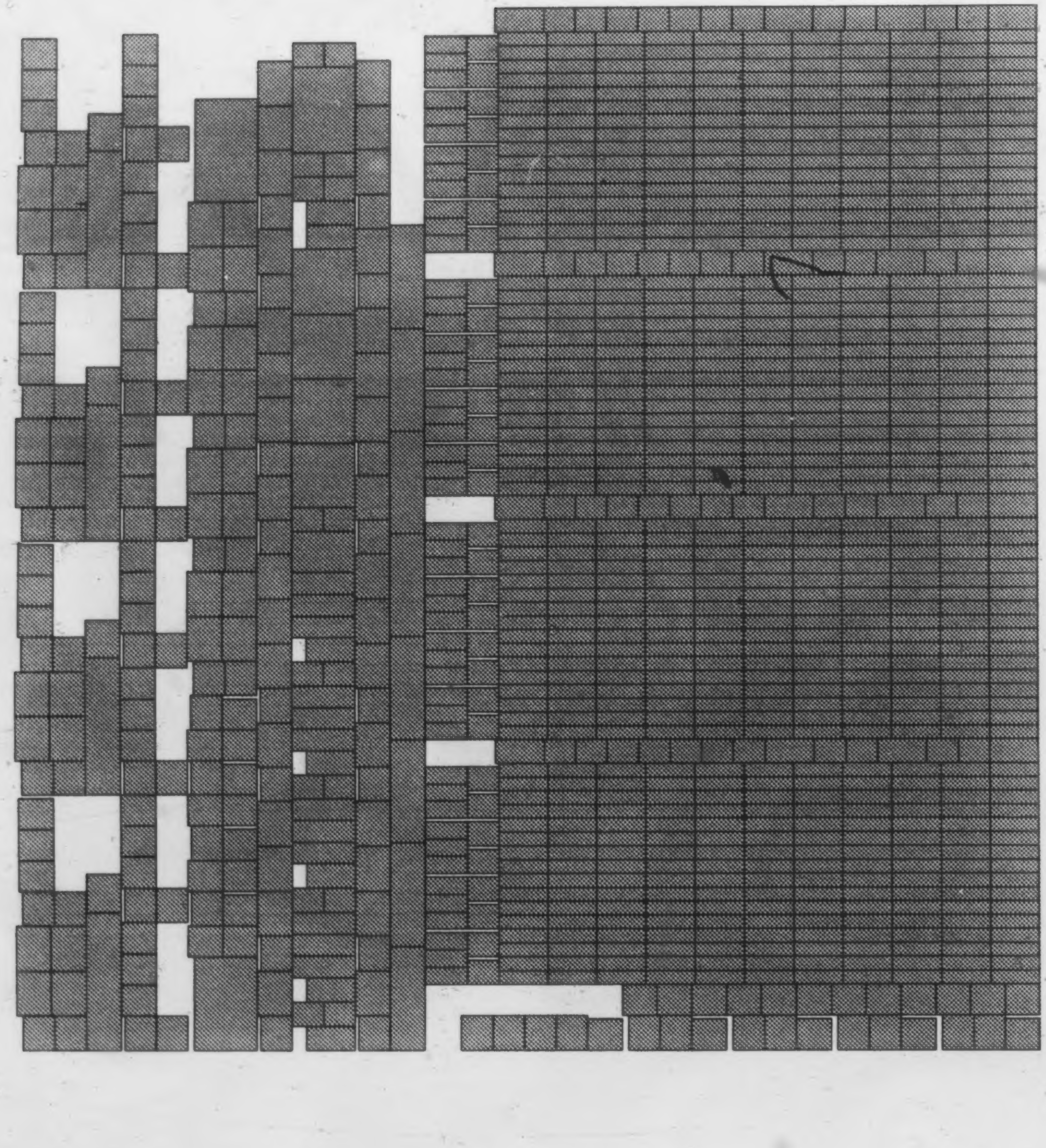


Figure B.2: Layout of Test Two

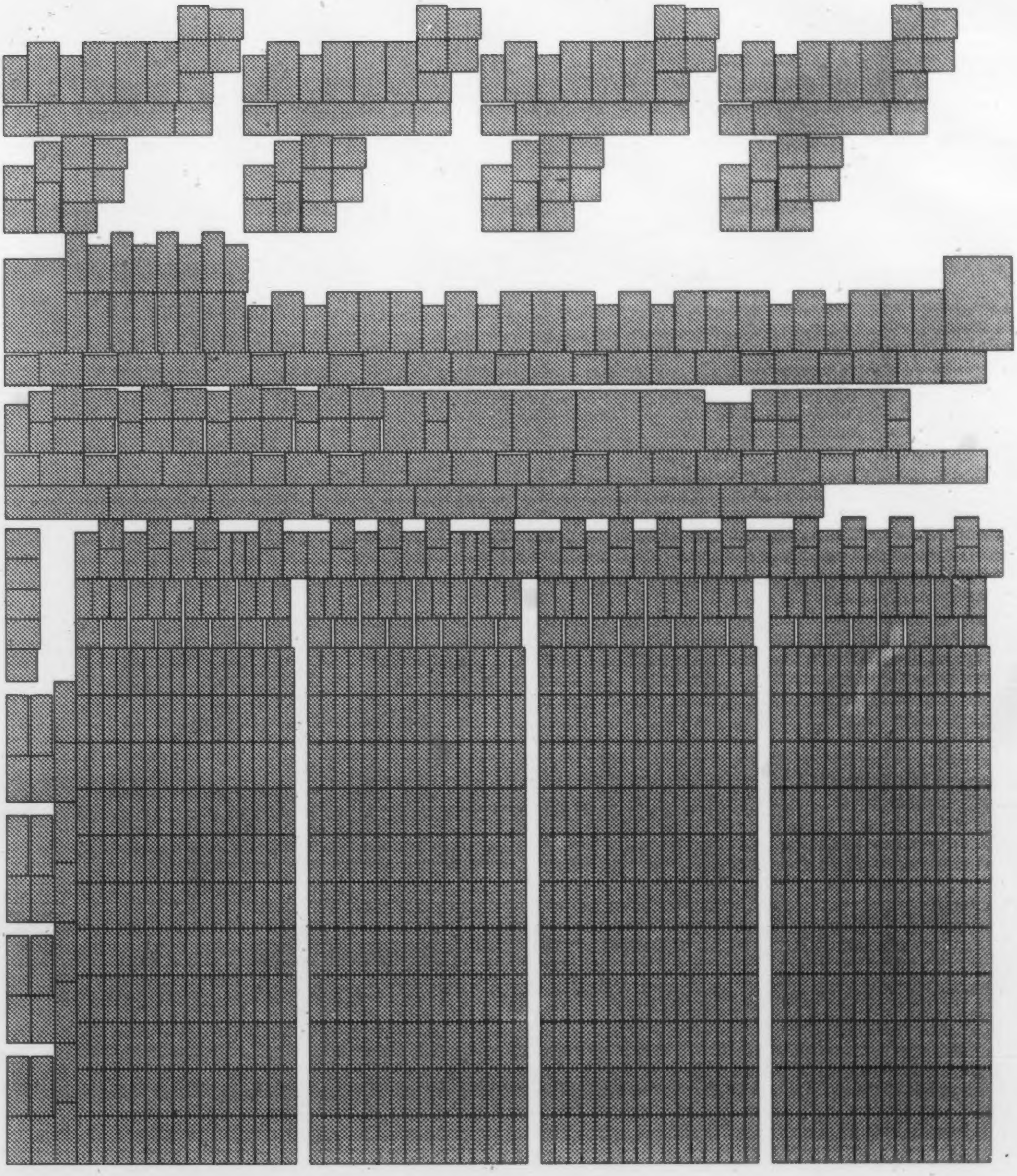
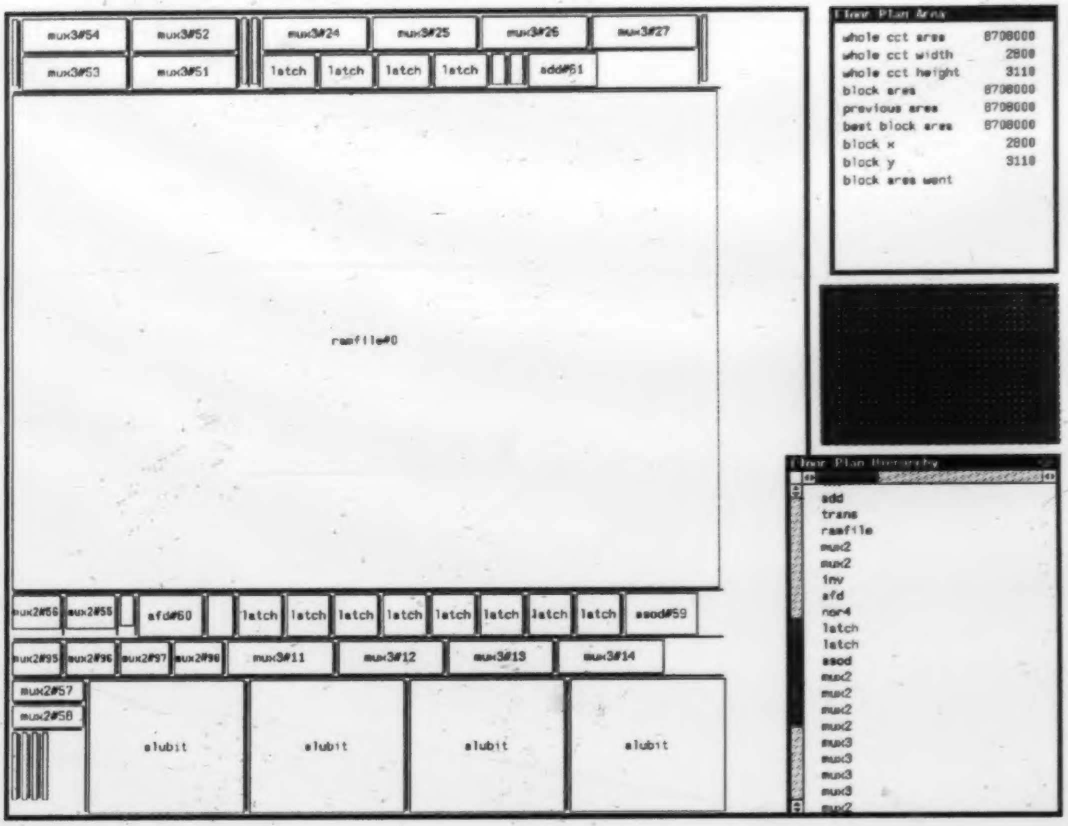


Figure B.3: Layout of Test Three


mux3#54
mux3#52
mux3#24
mux3#25
mux3#26
mux3#27
mux3#53
mux3#51
latch
latch
latch
latch
add#61
ramfile#0
mux2#56
mux2#55
afd#60
latch
latch
latch
latch
latch
latch
latch
latch
asod#59
mux2#95
mux2#96
mux2#97
mux2#98
mux3#11
mux3#12
mux3#13
mux3#14
mux2#57
mux2#58
alubit
alubit
alubit
alubit
Floor Plan Area
Floor Plan Hierarchy

```
whole cct area      8708000
whole cct width        2800
whole cct height       3110
block area          8708000
previous area       8708000
best block area     8708000
block x                2800
block y                3110
block area went
```

```
add
trans
ramfile
mux2
mux2
inv
afd
nor4
latch
latch
asod
mux2
mux2
mux2
mux2
mux3
mux3
mux3
mux3
mux2
```

Figure C.1: Screen Image of Final Floorplan of Test One